Eloquent JavaScript

3rd Edition

Marijn Haverbeke



目錄

JavaScript 编程精解 中文第三版

原书：Eloquent JavaScript 3rd edition

译者：飞龙

自豪地采用谷歌翻译

部分参考了《JavaScript 编程精解（第 2 版）》

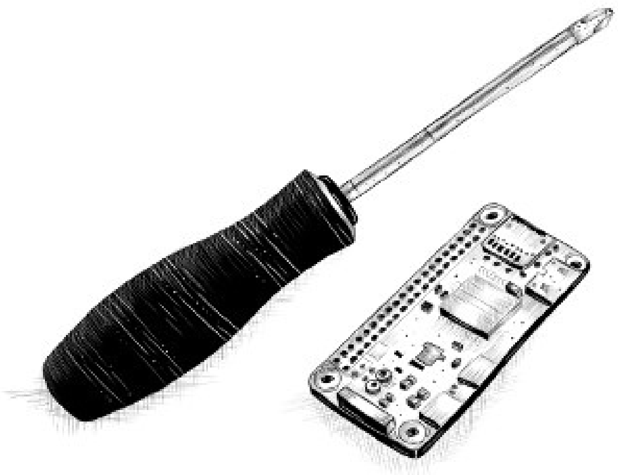
- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- 代码仓库

赞助我

协议

零、前言

> 原文：Introduction
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> We think we are creating the system for our own purposes. We believe we are making it in our own image... But the computer is not really like us. It is a projection of a very slim part of ourselves: that portion devoted to logic, order, rule, and clarity.
>
> Ellen Ullman，《Close to the Machine: Technophilia and its Discontents》



这是一本关于指导电脑的书。时至今日，计算机就像螺丝刀一样随处可见，但相比于螺丝刀而言，计算机更复杂一些，并且，让他们做你想让他们做的事情，并不总是那么容易。

如果你让计算机执行的任务是常见的，易于理解的任务，例如向你显示你的电子邮件，或像计算器一样工作，则可以打开相应的应用并开始工作。 但对于独特的或开放式的任务，应用可能不存在。

这就是编程可能出现的地方。编程是构建一个程序的行为 - 它是一组精确的指令，告诉计算机做什么。 由于计算机是愚蠢的，迂腐的野兽，编程从根本上是乏味和令人沮丧的。

幸运的是，如果你可以克服这个事实，并且甚至可以享受愚蠢机器可以处理的严谨思维，那么编程可以是非常有益的。 它可以让你在几秒钟内完成手动操作。 这是一种方法，让你的电脑工具去做它以前做不到的事情。 它也提供了抽象思维的优秀练习。

大多数编程都是用编程语言完成的。 编程语言是一种人工构建的语言，用于指导计算机。 有趣的是，我们发现与电脑沟通的最有效的方式，与我们彼此沟通的方式相差太大。 与人类语言一样，计算机语言可以以新的方式组合词语和词组，从而可以表达新的概念。

在某种程度上，基于语言的界面，例如 80 年代和 90 年代的 BASIC 和 DOS 提示符，是与计算机交互的主要方法。 这些已经在很大程度上被视觉界面取代，这些视觉界面更容易学习，但提供更少的自由。 计算机语言仍然存在，如果你知道在哪里看到。 每种现代 Web 浏览器都内置了一种这样的语言，即 JavaScript，因此几乎可以在所有设备上使用。

本书将试图让你足够了解这门语言，从而完成有用和有趣的东西。

关于程序设计

除了讲解 JavaScript 之外，本书也会介绍一些程序设计的基本原则。程序设计还是比较复杂的。编程的基本规则简单清晰，但在这些基本规则之上构建的程序却容易变得复杂，导致程序产生了自己的规则和复杂性。即便程序是按照你自己的思路去构建的，你也有可能迷失在代码之间。

在阅读本书时，你有可能会觉得书中的概念难以理解。如果你刚刚开始学习编程，那么你估计还有不少东西需要掌握呢。如果你想将所学知识融会贯通，那么就需要去多参考和学习一些资料。

是否付出必要的努力完全取决于你自己。当你阅读本书的时候发现任何难点，千万不要轻易就对自己的能力下结论。只要能坚持下去，你就是好样的。稍做休息，复习一下所学的知识点，始终确保自己阅读并理解了示例程序和相关的练习。学习是一项艰巨的任务，但你掌握的所有知识都属于你自己，而且今后的学习道路会愈加轻松。

> 当行动无利可图时，就收集信息；当信息无利可图时，就休息。
>
> Ursula K. Le Guin，《The Left Hand of Darkness》

一个程序有很多含义：它是开发人员编写的一段文本、计算机执行的一段指令集合、计算机内存当中的数据以及控制内存中数据的操作集合。我们通常很难将程序与我们日常生活中熟悉的事物进行对比。有一种表面上比较恰当的比喻，即将程序视作包含许多组件的机器，为了让机器正常工作，这些组件通过内部通信来实现整个机器的正常运转。

计算机是一台物理机器，充当这些非物质机器的载体。计算机本身并不能实现多么复杂的功能，但计算机之所以有用是因为它们的运算速度非常快。而程序的作用就是将这些看似简单的动作组合起来，然后实现复杂的功能。

程序是思想的结晶。编写程序不需要什么物质投入，它很轻量级，通过我们的双手创造。

但如果不稍加注意，程序的体积和复杂度就会失去控制，甚至代码的编写者也会感到迷惑。在可控的范围内编写程序是编程过程中首要解决的问题。当程序运行时，一切都是那么美好。编程的精粹就在于如何更好地控制复杂度。质量高的程序的复杂度都不会太高。

很多开发人员认为，控制程序复杂度的最好方法就是避免使用不熟悉的技术。他们制定了严格的规则（“最佳实践”），并小心翼翼地呆在他们安全区内。

这不仅无聊，而且也是无效的。新问题往往需要新的解决方案。编程领域还很年轻，仍然在迅速发展，并且多样到足以为各种不同的方法留出空间。在程序设计中有许多可怕的错误，你应该继续犯错，以便你能理解它们。好的程序看起来是什么样的感觉，是在实践中发展的，而不是从一系列规则中学到的。

为什么编程语言重要

在计算技术发展伊始，并没有编程语言这个概念。程序看起来就像这样：

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

该程序计算数字 1~10 之和，并打印出结果：`1+2+...+10=55`。该程序可以运行在一个简单的机器上。在早期计算机上编程时，我们需要在正确的位置设置大量开关阵列，或在纸带上穿孔并将纸带输入计算机中。你可以想象这个过程是多么冗长乏味且易于出错。即便是编写非常简单的程序，也需要有经验的人耗费很大精力才能完成。编写复杂的程序则更是难上加难。

当然了，手动输入这些晦涩难懂的位序列（1 和 0）来编写程序的确能让程序员感到很有成就感，而且能给你的职业带来极大的满足感。

在上面的程序中，每行都包含一条指令。我们可以用中文来描述这些指令：

1. 将数字 0 存储在内存地址中的位置 0。
2. 将数字 1 存储在内存地址的位置 1。
3. 将内存地址的位置 1 中的值存储在内存地址的位置 2。
4. 将内存地址的位置 2 中的值减去数字 11。
5. 如果内存地址的位置 2 中的值是 0，则跳转到指令 9。
6. 将内存地址的位置 1 中的值加到内存地址的位置 0。

7. 将内存地址的位置 1 中的值加上数字 1。
8. 跳转到指令 3。
9. 输出内存地址的位置 0 中的值。

虽说这已经比一大堆位序列要好读了许多，但仍然不清晰。使用名称而不是数字用于指令和存储位置有所帮助：

```
 Set “total” to 0.
 Set “count” to 1.
[loop]
 Set “compare” to “count”.
 Subtract 11 from “compare”.
 If “compare” is zero, continue at [end].
 Add “count” to “total”.
 Add 1 to “count”.
 Continue at [loop].
[end]
 Output “total”.
```

现在你能看出该程序是如何工作的吗？前两行代码初始化两个内存位置的值：`total` 用于保存累加计算结果，而 `count` 则用于记录当前数字。你可能觉得 `compare` 的那行代码看起来有些奇怪。程序想根据 `count` 是否等于 11 来决定是否应该停止运行。因为我们的机器相当原始，所以只能测试一个数字是否为 0，并根据它做出决策。因此程序用名为 `compare` 的内存位置存放 `count-11` 的值，并根据该值是否为 0 决定是否跳转。接下来两行将 `count` 的值累加到结果上，并将 `count` 加 1，直到 `count` 等于 `11` 为止。

下面使用 JavaScript 重新编写了上面的程序：

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
// → 55
```

这个版本的程序得到了一些改进。更为重要的是，我们再也不需要指定程序如何来回跳转了，而是由 `while` 结构负责完成这个任务。只要我们给予的条件成立，`while` 语句就会不停地执行其下方的语句块（包裹在大括号中）。而我们给予的条件是 `count<=10` ，意思是“`count` 小于等于 10”。我们再也不需要创建临时的值并将其与 0 比较，那样的代码十分烦琐。编程语言的一项职责就是，能够帮助我们处理这些烦琐无趣的逻辑。

在程序的结尾，也就是 `while` 语句结束后，我们使用 `console.log` 操作来输出结果。

最后，我们恰好有 `range` 和 `sum` 这类方便的操作。下面代码中的 `range` 函数用于创建数字集合，`sum` 函数用于计算数字集合之和：

```
console.log(sum(range(1, 10)));
// → 55
```

我们可以从这里了解到，同一个程序的长度可长可短，可读性可高可低。第一个版本的程序晦涩难懂，而最后一个版本的程序则接近于人类语言的表达方式：将 1~10 范围内的数字之和记录下来（我们会在后面的章节中详细介绍如何编写 `sum` 和 `range` 这样的函数）。

优秀的编程语言可以为开发人员提供更高层次的抽象，使用类似于人类语言的方式来与计算机进行交互。它有助于省略细节，提供便捷的积木（比如 `while` 和 `console.log` ），允许你定义自己的积木（比如 `sum` 和 `range` 函数），并使这些积木易于编写。。

什么是 JavaScript

JavaScript 诞生于 1995 年。起初，Netscape Navigator 浏览器将其运用在网页上添加程序。自此以后，各类主流图形网页浏览器均采用了 JavaScript。JavaScript 使得现代网页应用程序成为可能 —— 使用 JavaScript 可以直接与用户交互，从而避免每一个动作都需要重新载入页面。但有许多传统网站也会使用 JavaScript 来提供实时交互以及更加智能的表单功能。

JavaScript 其实和名为Java的程序设计语言没有任何关系。起了这么一个相似的名字完全是市场考虑使然，这并非是一个明智的决定。当 JavaScript 出现时，Java 语言已在市场上得到大力推广且拥有了极高人气，因此某些人觉得依附于 Java 的成功是个不错的主意。而我们现在已经无法摆脱这个名字了。

在 JavaScript 被广泛采用之后，ECMA 国际制订了一份标准文档来描述 JavaScript 的工作行为，以便所有声称支持 JavaScript 的软件都使用同一种语言。标准化完成后，该标准被称为 ECMAScript 标准。实际上，术语 ECMAScript 和 JavaScript 可以交换使用。它们不过是同一种语言的两个名字而已。

许多人会说 JavaScript 语言的坏话。这其中有很多这样的言论都是正确的。当被要求第一次使用 JavaScript 编写代码时，我当时就觉得这门语言难以驾驭。JavaScript 接受我输入的任何代码，但是又使用和我的想法完全不同的方式来解释代码。由于我没有任何线索知道我之前做了什么，因此我需要做出更多工作，但这也就存在一个实际问题：我们可以自由使用 JavaScript，而这种自由却几乎没有限度。这种设计其实是希望初学者更容易使用 JavaScript 编写程序。但实际上，系统不会指出我们错在何处，因此从程序中找出问题变得更加棘手。

但这种自由性也有其优势，许多技术在更为严格的语言中不可能实现，而在 JavaScript 中则留下了实现的余地，正如你看到的那样（比如第十章），有些优势可以弥补 JavaScript 的一些缺点。在正确地学习 JavaScript 并使用它工作了一段时间后，我真正喜欢上了 JavaScript。

JavaScript 版本众多。大约在 2000~2010 年间，这正是 JavaScript 飞速发展的时期，浏览器支持最多的是 ECMAScript 3。在此期间，ECMA 着手制定 ECMAScript 4，这是一个雄心勃勃的版本，ECMA 计划在这个版本中加入许多彻底的改进与扩展。但由于 ECMAScript 3 被

广泛使用，这种过于激进的修改必然会遭遇重重阻碍，最后 ECMA 不得不于 2008 年放弃了版本 4 的制定。这就产生了不那么雄心勃勃的版本 5，这只是一些没有争议的改进，出现在 2009 年。 然后版本 6 在 2015 年诞生，这是一个重大的更新，其中包括计划用于版本 4 的一些想法。从那以后，每年都会有新的更新。

语言不断发展的事实意味着，浏览器必须不断跟上，如果你使用的是较老的浏览器，它可能不支持每个特性。 语言设计师会注意，不要做任何可能破坏现有程序的改变，所以新的浏览器仍然可以运行旧的程序。 在本书中，我使用的是 2017 版的 JavaScript。

Web 浏览器并不是唯一一个可以运行 JavaScript 的平台。有些数据库，比如 MongoDB 和 CouchDB，也使用 JavaScript 作为脚本语言和查询语言。一些桌面和服务器开发的平台，特别是 Node.js 项目（第二十章介绍），为浏览器以外的 JavaScript 编程提供了一个环境。

代码及相关工作

代码是程序的文本内容。本书多数章节都介绍了大量代码。我相信阅读代码和编写代码是学习编程不可或缺的部分。尝试不要仅仅看一眼示例，而应该认真阅读并理解每个示例。刚开始使用这种方式可能会速度较慢并为代码所困惑，但我坚信你很快就可以熟能生巧。对待习题的方法也应该一样。除非你确实已经编写代码解决了问题，否则不要假设你已经理解了问题。

建议读者应尝试在实际的 JavaScript 解释器中执行习题代码。这样一来，你就可以马上获知代码工作情况的反馈，而且我希望读者去做更多的试验，而不仅仅局限于习题的要求。

可以在 http://eloquentjavascript.net/ 中查阅本书的在线版本，并运行和实验本书中的代码。也可以在在线版本中点击任何代码示例来编辑、运行并查看其产生的输出。在做习题时，你可以访问 http://eloquentjavascript.net/code/，该网址会提供每个习题的初始代码，让你专心于解答习题。

如果想要在本书提供的沙箱以外执行本书代码，需要稍加注意。许多的示例是独立的，而且可以在任何 JavaScript 环境下运行。但后续章节的代码大多数都是为特定环境（浏览器或者 Node.js）编写的，而且只能在这些特定环境下执行代码。此外，许多章节定义了更大的程序，这些章节中出现的代码片段会互相依赖或是依赖于一些外部文件。本书网站的沙箱提供了 zip 压缩文件的链接，该文件包含了所有运行特定章节代码所需的脚本和数据文件。

本书概览

本书包括三个部分。前十二章讨论 JavaScript 语言本身的一些特性。接下来的 8 章讨论网页浏览器和 JavaScript 在网页编程中的实践。最后两章专门讲解另一个使用 JavaScript 编程的环境 —— Node.js。

纵观本书，共有 5 个项目实战章，用于讲解规模较大的示例程序，你可以通过这些章来仔细品味真实的编程过程。根据项目出现次序，我们会陆续构建递送机器人（7）、程序设计语言（12）、平台游戏（16）、像素绘图程序（19）和一个动态网站（21）。

本书介绍编程语言时，首先使用4章来介绍 JavaScript 语言的基本结构，包括第二章控制结构（比如在本前言中看到的 `while` 单词）、第三章函数（编写你自己的积木）和第四章数据结构。此后你就可以编写简单的程序了。接下来，第五章和第六章介绍函数和对象的运用技术，以编写更加抽象的代码并以此来控制复杂度。

介绍完第一个项目实战（7）之后，将会继续讲解语言部分，例如第八章错误处理和 bug 修复、第九章正则表达式（处理文本数据的重要工具）、第十章模块化（解决复杂度的问题）以及第十一章异步编程（处理需要时间的事件）。第二个项目实战章节（12）则是对本书第一部分的总结。

第二部分（第十三章到第十九章），阐述了浏览器 JavaScript 中的一些工具。你将会学到在屏幕上显示某些元素的方法（第十四章与第十七章），响应用户输入的方法（第十五章）和通过网络通信的方法（第十八章）。这部分又有两个项目实战章节。

此后，第二十章阐述 Node.js，而第二十一章使用该工具构建一个简单的网页系统。

本书版式约定

本书中存在大量代码，程序（包括你迄今为止看到的一些示例）代码的字体如下所示：

```
function factorial(n) {
  if (n == 0) {
    return 1;
  } else {
    return factorial(n - 1) * n;
  }
}
```

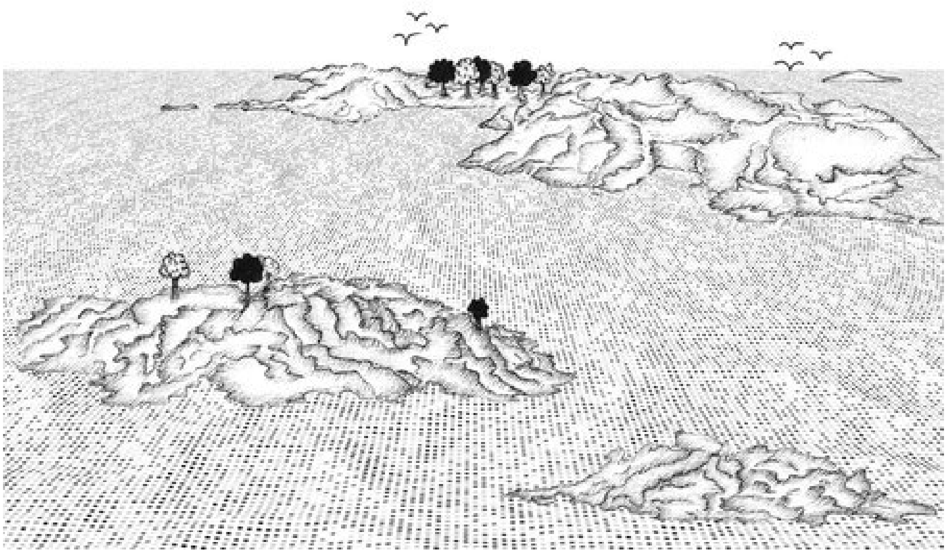
为了展示程序产生的输出，本书常在代码后编写代码期望输出，输出结果前会加上两个反斜杠和一个箭头。

```
console.log(factorial(8));
// → 40320
```

祝好运！

一、值，类型和运算符

> 原文：Values, Types, and Operators
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 在机器的表面之下，程序在运转。 它不费力就可以扩大和缩小。 在和谐的关系中，电子散开并重新聚合。 监视器上的表格只是水面上的涟漪。 本质隐藏在下面。
>
> Master Yuan-Ma，《The Book of Programming》



计算机世界里只有数据。 你可以读取数据，修改数据，创建新数据 - 但不能提及不是数据的东西。 所有这些数据都以位的长序列存储，因此基本相似。

位是任何类型的二值的东西，通常描述为零和一。 在计算机内部，他们有一些形式，例如高电荷或低电荷，强信号或弱信号，或 CD 表面上的亮斑点或暗斑点。 任何一段离散信息都可以简化为零和一的序列，从而以位表示。

例如，我们可以用位来表示数字 13。 它的原理与十进制数字相同，但不是 10 个不同的数字，而只有 2 个，每个数字的权重从右到左增加 2 倍。 以下是组成数字 13 的位，下方显示数字的权重：

```
   0   0   0   0   1   1   0   1
 128  64  32  16   8   4   2   1
```

因此，这就是二进制数`00001101`，或者`8+4+1`，即 13。

值

想象一下位之海 - 一片它们的海洋。 典型的现代计算机的易失性数据存储器（工作存储器）中，有超过 300 亿位。非易失性存储（硬盘或等价物）往往还有几个数量级。

为了能够在不丢失的情况下，处理这些数量的数据，我们必须将它们分成代表信息片段的块。 在 JavaScript 环境中，这些块称为值。 虽然所有值都是由位构成的，但他们起到不同的作用，每个值都有一个决定其作用的类型。 有些值是数字，有些值是文本片段，有些值是函数，等等。

要创建一个值，你只需要调用它的名字。 这很方便。 你不必为你的值收集建筑材料或为其付费。 你只需要调用它，然后刷的一下，你就有了它。 当然，它们并不是真正凭空创造的。 每个值都必须存储在某个地方，如果你想同时使用大量的值，则可能会耗尽内存。 幸运的是，只有同时需要它们时，这才是一个问题。 只要你不再使用值，它就会消失，留下它的一部分作为下一代值的建筑材料。

本章将会介绍 JavaScript 程序当中的基本元素，包括简单的值类型以及值运算符。

数字

数字（`Number`）类型的值即数字值。在 JavaScript 中写成如下形式：

```
13
```

在程序中使用这个值的时候，就会将数字 13 以位序列的方式存放在计算机的内存当中。

JavaScript使用固定数量的位（64 位）来存储单个数字值。 你可以用 64 位创造很多模式，这意味着可以表示的不同数值是有限的。 对于`N`个十进制数字，可以表示的数值数量是`10^N`。 与之类似，给定 64 个二进制数字，你可以表示`2^64`个不同的数字，大约 18 亿亿（18 后面有 18 个零）。太多了。

过去计算机内存很小，人们倾向于使用一组 8 位或 16 位来表示他们的数字。 这么小的数字很容易意外地溢出，最终得到的数字不能放在给定的位数中。 今天，即使是装在口袋里的电脑也有足够的内存，所以你可以自由使用 64 位的块，只有在处理真正的天文数字时才需要担心溢出。

不过，并非所有 18 亿亿以下的整数都能放在 JavaScript 数值中。 这些位也存储负数，所以一位用于表示数字的符号。 一个更大的问题是，也必须表示非整数。 为此，一些位用于存储小数点的位置。 可以存储的实际最大整数更多地在 9 亿亿（15 个零）的范围内 - 这仍然相当多。

使用小数点来表示分数。

```
9.81
```

对于非常大或非常小的数字，你也可以通过输入`e`（表示指数），后面跟着指数来使用科学记数法：

```
2.998e8
```

即`2.998 * 10^8 = 299,800,000`。

当计算小于前文当中提到的 9000 万亿的整数时，其计算结果会十分精确，不过在计算小数的时候精度却不高。正如（`pi`）无法使用有限个数的十进制数字表示一样，在使用 64 位来存储分数时也同样会丢失一些精度。虽说如此，但这类丢失精度只会在一些特殊情况下才会出现问题。因此我们需要注意在处理分数时，将其视为近似值，而非精确值。

算术

与数字密切相关的就是算术。比如，加法或者乘法之类的算术运算会使用两个数值，并产生一个新的数字。JavaScript 中的算术运算如下所示：

```
100 + 4 * 11
```

我们把`+`和`*`符号称为运算符。第一个符号表示加法，第二个符号表示乘法。将一个运算符放在两个值之间，该运算符将会使用其旁边的两个值产生一个新值。

但是这个例子的意思是“将 4 和 100 相加，并将结果乘 11”，还是是在加法之前计算乘法？正如你可能猜到的那样，乘法首先计算。但是和数学一样，你可以通过将加法包在圆括号中来改变它：

```
(100 + 4) * 11
```

`-`运算符表示减法，`/`运算符则表示除法。

在运算符同时出现，并且没有括号的情况下，其运算顺序根据运算符优先级确定。示例中的乘法运算符优先级高于加法。而`/`运算符和`*`运算符优先级相同，`+`运算符和`-`运算符优先级也相同。当多个具有相同优先级的运算符相邻出现时，运算从左向右执行，比如`1-2+1`的运算顺序是`(1-2)+1`。

你无需担心这些运算符的优先级规则，不确定的时候只需要添加括号即可。

还有一个算术运算符，你可能无法立即认出。 `%` 符号用于表示取余操作。`X % Y` 是 `Y` 除 `X` 的余数。 例如，`314 % 100` 产生 `14` ，`144 % 12` 产生 `0` 。 余数的优先级与乘法和除法的优先级相同。 你还经常会看到这个运算符被称为模运算符。

特殊数字

在 JavaScript 中有三个特殊的值，它们虽然是数字，但看起来却跟一般的数字不太一样。

前两个是 `Infinity` 和 `-Infinity` ，它们代表正无穷和负无穷。“无穷减一”仍然是“无穷”，依此类推。 尽管如此，不要过分信任基于无穷大的计算。 它在数学上不合理，并且很快导致我们的下一个特殊数字： `NaN` 。

`NaN` 代表“不是数字”，即使它是数字类型的值。 例如，当你尝试计算 `0/0` （零除零）， `Infinity - Infinity` 或任何其他数字操作，它不会产生有意义的结果时，你将得到此结果。

字符串

下一个基本数据类型是字符串（ `String` ）。 字符串用于表示文本。 它们是用引号括起来的：

```
`Down on the sea`
"Lie on the ocean"
'Float on the ocean'
```

只要字符串开头和结尾的引号匹配，就可以使用单引号，双引号或反引号来标记字符串。

几乎所有的东西都可以放在引号之间，并且 JavaScript 会从中提取字符串值。 但少数字符更难。 你可能难以想象，如何在引号之间加引号。 当使用反引号（ `` ` `` ）引用字符串时，换行符（当你按回车键时获得的字符）可能会被包含，而无需转义。

若要将这些字符存入字符串，需要使用下列规则：当反斜杠（ `\` ）出现在引号之间的文本中时，表示紧跟在其后的字符具有特殊含义，我们将其称之为转义符。当引号紧跟在反斜杠后时，并不意味着字符串结束，而表示这个引号是字符串的一部分。当字符 `n` 出现在反斜杠后时，JavaScript 将其解释成换行符。以此类推， `\t` 表示制表符，我们来看看下面这个字符串：

```
"This is the first line\nAnd this is the second"
```

该字符串实际表示的文本是：

```
This is the first line
And this is the second
```

当然，在某些情况下，你希望字符串中的反斜杠只是反斜杠，而不是特殊代码。 如果两个反斜杠写在一起，它们将合并，并且只有一个将留在结果字符串值中。 这就是字符串“`A newline character is written like "\n".`”的表示方式：

```
"A newline character is written like \"\\n\"."
```

字符串也必须建模为一系列位，以便能够存在于计算机内部。 JavaScript 执行此操作的方式基于 Unicode 标准。 该标准为你几乎需要的每个字符分配一个数字，包括来自希腊语，阿拉伯语，日语，亚美尼亚语，以及其他的字符。 如果我们为每个字符分配一个数字，则可以用一系列数字来描述一个字符串。

这就是 JavaScript 所做的。 但是有一个复杂的问题：JavaScript 的表示为每个字符串元素使用 16 位，它可以描述多达 2 的 16 次方个不同的字符。 但是，Unicode 定义的字符多于此 - 大约是此处的两倍。 所以有些字符，比如许多 emoji，在 JavaScript 字符串中占据了两个“字符位置”。 我们将在第 5 章中回来讨论。

我们不能将除法，乘法或减法运算符用于字符串，但是`+`运算符却可以。这种情况下，运算符并不表示加法，而是连接操作：将两个字符串连接到一起。以下语句可以产生字符串`"concatenate"`：

```
"con" + "cat" + "e" + "nate"
```

字符串值有许多相关的函数（方法），可用于对它们执行其他操作。 我们将在第 4 章中回来讨论。

用单引号或双引号编写的字符串的行为非常相似 - 唯一的区别是需要在其中转义哪种类型的引号。 反引号字符串，通常称为模板字面值，可以实现更多的技巧。 除了能够跨越行之外，它们还可以嵌入其他值。

```
`half of 100 is ${100 / 2}`
```

当你在模板字面值中的`$ {}`中写入内容时，将计算其结果，转换为字符串并包含在该位置。 这个例子产生`"half of 100 is 50"`。

一元运算符

并非所有的运算符都是用符号来表示，还有一些运算符是用单词表示的。比如`typeof`运算符，会产生一个字符串的值，内容是给定值的具体类型。

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

我们将在示例代码中使用`console.log`，来表示我们希望看到求值结果。更多内容请见下一章。

我们所见过的绝大多数运算符都使用两个值进行操作，而`typeof`仅接受一个值进行操作。使用两个值的运算符称为二元运算符，而使用一个值的则称为一元运算符。减号运算符既可用作一元运算符，也可用作二元运算符。

```
console.log(- (10 - 2))
// → -8
```

布尔值

拥有一个值，它能区分两种可能性，通常是有用的，例如“是”和“否”或“开”和“关”。 为此，JavaScript 拥有布尔（`Boolean`）类型，它有两个值：`true`和`false`，它们就写成这些单词。

比较

一种产生布尔值的方法如下所示：

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

`>`和`<`符号分别表示“大于”和“小于”。这两个符号是二元运算符，通过该运算符返回的结果是一个布尔值，表示其运算是否为真。

我们可以使用相同的方法比较字符串。

```
console.log("Aardvark" < "Zoroaster")
// → true
```

字符串排序的方式大致是字典序，但不真正是你期望从字典中看到的那样：大写字母总是比小写字母“小”，所以`"Z"<"A"`，非字母字符（`!`，`-`等）也包含在排序中。 比较字符串时，JavaScript 从左向右遍历字符，逐个比较 Unicode 代码。

其他类似的运算符则包括`>=`（大于等于），`<=`（小于等于），`==`（等于）和`!=`（不等于）。

```
console.log("Apple" == "Orange")
// → false
```

在 JavaScript 中，只有一个值不等于其自身，那就是`NaN`（Not a Number，非数值）。

```
console.log(NaN == NaN)
// → false
```

`NaN` 用于表示非法运算的结果，正因如此，不同的非法运算结果也不会相等。

逻辑运算符

还有一些运算符可以应用于布尔值上。JavaScript 支持三种逻辑运算符：与（and），或（or）和非（not）。这些运算符可以用于推理布尔值。

`&&` 运算符表示逻辑与，该运算符是二元运算符，只有当赋给它的两个值均为 `true` 时其结果才是真。

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

`||` 运算符表示逻辑或。当两个值中任意一个为 `true` 时，结果就为真。

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

感叹号（`!`）表示逻辑非，该运算符是一元运算符，用于反转给定的值，比如 `!true` 的结果是 `false`，而 `!false` 结果是 `true`。

在混合使用布尔运算符和其他运算符的情况下，总是很难确定什么时候需要使用括号。实际上，只要熟悉了目前为止我们介绍的运算符，这个问题就不难解决了。`||` 优先级最低，其次是 `&&`，接着是比较运算符（`>`，`==` 等），最后是其他运算符。基于这些优先级顺序，我们在一般情况下最好还是尽量少用括号，比如说：

```
1 + 1 == 2 && 10 * 10 > 50
```

现在我们来讨论最后一个逻辑运算符，它既不属于一元运算符，也不属于二元运算符，而是三元运算符（同时操作三个值）。该运算符由一个问号和冒号组成，如下所示。

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

这个被称为条件运算符（或者有时候只是三元运算符，因为它是该语言中唯一的这样的运算符）。问号左侧的值“挑选”另外两个值中的一个。当它为真，它选择中间的值，当它为假，则是右边的值。

空值

有两个特殊值，写成 `null` 和 `undefined`，用于表示不存在有意义的值。它们本身就是值，但它们没有任何信息。

在 JavaScript 语言中，有许多操作都会产生无意义的值（我们会在后面的内容中看到实例），这些操作会得到 `undefined` 的结果仅仅只是因为每个操作都必须产生一个值。

`undefined` 和 `null` 之间的意义差异是 JavaScript 设计的一个意外，大多数时候它并不重要。在你实际上不得不关注这些值的情况下，我建议将它们视为几乎可互换的。

自动类型转换

在引言中，我提到 JavaScript 会尽可能接受几乎所有你给他的程序，甚至是那些做些奇怪事情的程序。以下表达式很好地证明了这一点：

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

当运算符应用于类型“错误”的值时，JavaScript 会悄悄地将该值转换为所需的类型，并使用一组通常不是你想要或期望的规则。这称为类型转换。第一个表达式中的 `null` 变为 `0`，第二个表达式中的 `"5"` 变为 `5`（从字符串到数字）。然而在第三个表达式中，`+` 在数字加法之前尝试字符串连接，所以 `1` 被转换为 `"1"`（从数字到字符串）。

当某些不能明显映射为数字的东西（如 `"five"` 或 `undefined`）转换为数字时，你会得到值 `NaN`。`NaN` 进一步的算术运算会产生 `NaN`，所以如果你发现自己在一个意想不到的地方得到了它，需要寻找意外的类型转换。

当相同类型的值之间使用 `==` 符号进行比较时，其运算结果很好预测：除了 `NaN` 这种情况，只要两个值相同，则返回 `true`。但如果类型不同，JavaScript 则会使用一套复杂难懂的规则来确定输出结果。在绝大多数情况下，JavaScript 只是将其中一个值转换成另一个值的类型。但如果运算符两侧存在 `null` 或 `undefined`，那么只有两侧均为 `null` 或 `undefined` 时结果才为 `true`。

```
console.log(null == undefined);
// → true
console.log(null == 0);
// → false
```

这种行为通常很有用。 当你想测试一个值是否具有真值而不是`null`或`undefined`时，你可以用`==`（或`!=`）运算符将它与`null`进行比较。

但是如果你想测试某些东西是否严格为“false”呢？ 字符串和数字转换为布尔值的规则表明，`0`，`NaN`和空字符串（`""`）计为`false`，而其他所有值都计为`true`。 因此，像`'0 == false'`和`"" == false`这样的表达式也是真的。 当你不希望发生自动类型转换时，还有两个额外的运算符：`===`和`!==`。 第一个测试是否严格等于另一个值，第二个测试它是否不严格相等。 所以`"" === false`如预期那样是错误的。

我建议使用三字符比较运算符来防止意外类型转换的发生，避免作茧自缚。但如果比较运算符两侧的值类型是相同的，那么使用较短的运算符也没有问题。

逻辑运算符的短路特性

逻辑运算符`&&`和`||`以一种特殊的方式处理不同类型的值。 他们会将其左侧的值转换为布尔型，来决定要做什么，但根据运算符和转换结果，它们将返回原始的左侧值或右侧值。

例如，当左侧值可以转换为`true`时，`||`运算符会返回它，否则返回右侧值。 当值为布尔值时，这具有预期的效果，并且对其他类型的值做类似的操作。

```
console.log(null || "user")
// → user
console.log("Agnes" || "user")
// → Agnes
```

我们可以此功能用作回落到默认值的方式。 如果你的一个值可能是空的，你可以把`||`和备选值放在它之后。 如果初始值可以转换为`false`，那么你将得到备选值。

`&&`运算符工作方式与其相似但不相同。当左侧的值可以被转换成`false`时，`&&`运算符会返回左侧值，否则返回右侧值。

这两个运算符的另一个重要特性是，只在必要时求解其右侧的部分。 在`true || X`的情况下，不管`X`是什么 - 即使它是一个执行某些恶意操作的程序片段，结果都是`true`，并且`X`永远不会求值。 `false && X`也是一样，它是`false`的，并且忽略`X`。 这称为短路求值。

条件运算符以类似的方式工作。 在第二个和第三个值中，只有被选中的值才会求值。

本章小结

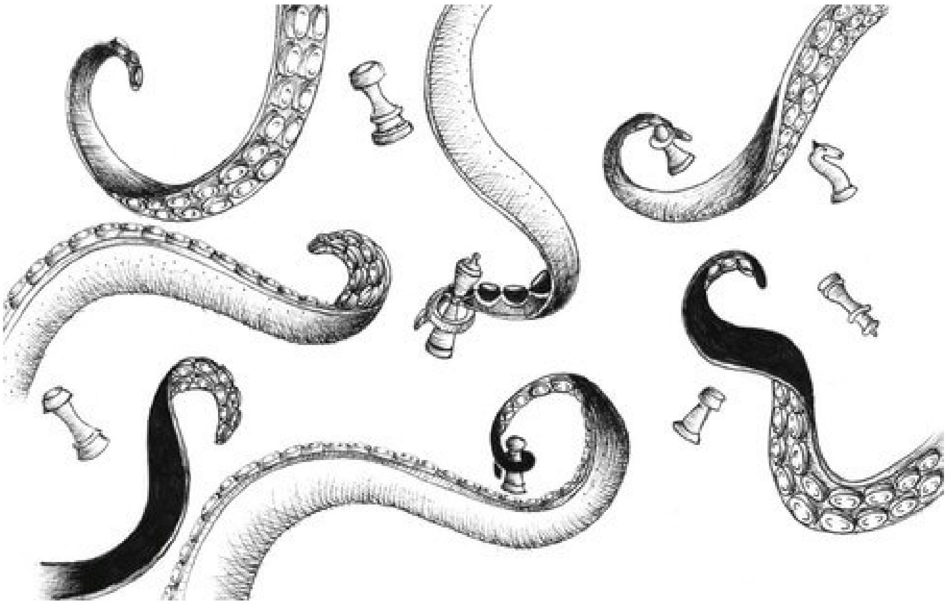
在本章中，我们介绍了 JavaScript 的四种类型的值：数字，字符串，布尔值和未定义值。

通过输入值的名称（`true`，`null`）或值（`13`，`"abc"`）就可以创建它们。你还可以通过运算符来对值进行合并和转换操作。本章已经介绍了算术二元运算符（`+`，`-`，`*`，`/`和`%`），字符串连接符（`+`），比较运算符（`==`，`!=`，`===`，`!==`，`<`，`>`，`<=`和`>=`），逻辑运算符（`&&`和`||`）和一些一元运算符（`-`表示负数，`!`表示逻辑非，`typeof`用于查询值的类型）。

这为你提供了足够的信息，将 JavaScript 用作便携式计算器，但并不多。 下一章将开始将这些表达式绑定到基本程序中。

二、程序结构

> 原文：Program Structure
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> And my heart glows bright red under my filmy, translucent skin and they have to administer 10cc of JavaScript to get me to come back. (I respond well to toxins in the blood.) Man, that stuff will kick the peaches right out your gills!
>
> why，《Why's (Poignant) Guide to Ruby》



在本章中，我们开始做一些实际上称为编程的事情。 我们将扩展我们对 JavaScript 语言的掌控，超出我们目前所看到的名词和句子片断，直到我们可以表达有意义的散文。

表达式和语句

在第 1 章中，我们为它们创建了值，并应用了运算符来获得新的值。 像这样创建值是任何 JavaScript 程序的主要内容。 但是，这种东西必须在更大的结构中构建，才能发挥作用。 这就是我们接下来要做的。

我们把产生值的操作的代码片段称为表达式。按照字面含义编写的值（比如 `22` 或 `"psychoanalysis"`）都是一个表达式。而括号当中的表达式、使用二元运算符连接的表达式或使用一元运算符的表达式，仍然都是表达式。

这展示了一部分基于语言的接口之美。 表达式可以包含其他表达式，其方式非常类似于人类语言的从句嵌套 - 从句可以包含它自己的从句，依此类推。 这允许我们构建描述任意复杂计算的表达式。

如果一个表达式对应一个句子片段，则 JavaScript 语句对应于一个完整的句子。 一个程序是一列语句。

最简单的一条语句由一个表达式和其后的分号组成。比如这就是一个程序：

```
1;
!false;
```

不过，这是一个无用的程序。 表达式可以仅仅满足于产生一个值，然后可以由闭合的代码使用。 一个声明是独立存在的，所以它只有在影响到世界的时候才会成立。 它可以在屏幕上显示某些东西 - 这可以改变世界 - 或者它可以改变机器的内部状态，从而影响后面的语句。 这些变化被称为副作用。 前面例子中的语句仅仅产生值 `1` 和 `true`，然后立即将它们扔掉。 这给世界没有留下什么印象。 当你运行这个程序时，什么都不会发生。

在某些情况下，JavaScript 允许您在语句结尾处省略分号。 在其他情况下，它必须在那里，否则下一行将被视为同一语句的一部分。 何时可以安全省略它的规则有点复杂且容易出错。 所以在本书中，每一个需要分号的语句都会有分号。 至少在你更了解省略分号的细节之前，我建议你也这样做。

绑定

程序如何保持内部状态？ 它如何记住东西？ 我们已经看到如何从旧值中产生新值，但这并没有改变旧值，新值必须立即使用，否则将会再度消失。 为了捕获和保存值，JavaScript 提供了一种称为绑定或变量的东西：

```
let caught = 5 * 5;
```

这是第二种语句。 关键字（keyword）`let` 表示这个句子打算定义一个绑定。 它后面跟着绑定的名称，如果我们想立即给它一个值，使用 `=` 运算符和一个表达式。

前面的语句创建一个名为 `caught` 的绑定，并用它来捕获乘以 `5 * 5` 所产生的数字。

在定义绑定之后，它的名称可以用作表达式。 这种表达式的值是绑定当前所持有的值。 这是一个例子：

```
let ten = 10;
console.log(ten * ten);
// → 100
```

当绑定指向某个值时，并不意味着它永远与该值绑定。 可以在现有的绑定上随时使用`=`运算符，将它们与当前值断开连接，并让它们指向一个新值：

```
var mood = "light";
console.log(mood);
// → light
mood = "dark";
console.log(mood);
// → dark
```

你应该将绑定想象为触手，而不是盒子。 他们不包含值; 他们捕获值 - 两个绑定可以引用相同的值。 程序只能访问它还在引用的值。 当你需要记住某些东西时，你需要长出一个触手来捕获它，或者你重新贴上你现有的触手之一。

我们来看另一个例子。 为了记住 Luigi 欠你的美元数量，你需要创建一个绑定。 然后当他还你 35 美元时，你赋予这个绑定一个新值：

```
let luigisDebt = 140;
luigisDebt = luigisDebt - 35;
console.log(luigisDebt);
// → 105
```

当你定义一个绑定而没有给它一个值时，触手没有任何东西可以捕获，所以它只能捕获空气。 如果你请求一个空绑定的值，你会得到`undefined`值。

一个`let`语句可以同时定义多个绑定，定义必需用逗号分隔。

```
let one = 1, two = 2;
console.log(one + two);
// → 3
```

`var`和`const`这两个词也可以用来创建绑定，类似于`let`。

```
var name = "Ayda";
const greeting = "Hello ";
console.log(greeting + name);
// → Hello Ayda
```

第一个`var`（“variable”的简写）是 JavaScript 2015 之前声明绑定的方式。 我们在下一章中，会讲到它与`let`的确切的不同之处。 现在，请记住它大部分都做同样的事情，但我们很少在本书中使用它，因为它有一些令人困惑的特性。

`const`这个词代表常量。 它定义了一个不变的绑定，只要它存在，它就指向相同的值。 这对于一些绑定很有用，它们向值提供一个名词，以便之后可以很容易地引用它。

绑定名称

绑定名称可以是任何单词。数字可以是绑定名称的一部分，例如`catch22`是一个有效的名称，但名称不能以数字开头。绑定名称可能包含美元符号（`$`）或下划线（`_`），但不包含其他标点符号或特殊字符。

具有特殊含义的词，如`let`，是关键字，它们不能用作绑定名称。在未来的 JavaScript 版本中还有一些“保留供使用”的单词，它们也不能用作绑定名称。关键字和保留字的完整列表相当长：

```
break case catch class const continue debugger default
delete do else enum export extends false finally for
function if implements import interface in instanceof let
new package private protected public return static super
switch this throw true try typeof var void while with yield
```

不要担心记住这些东西。创建绑定时会产生意外的语法错误，请查看您是否尝试定义保留字。

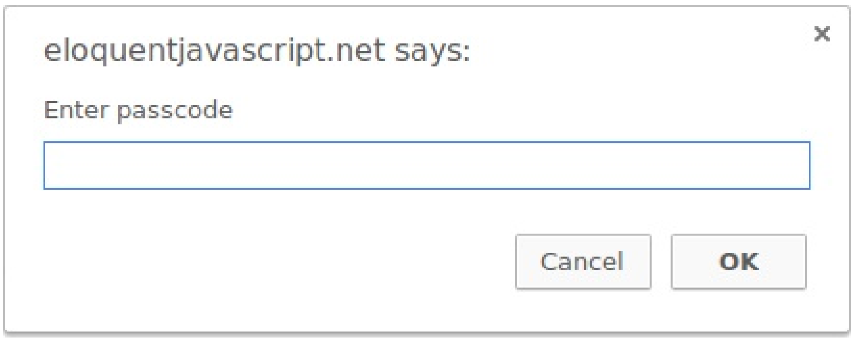
环境

给定时间中存在的绑定及其值的集合称为环境。当一个程序启动时，这个环境不是空的。它总是包含作为语言标准一部分的绑定，并且在大多数情况下，它还具有一些绑定，提供与周围系统交互的方式。例如，在浏览器中，有一些功函数能可以与当前加载的网站交互并读取鼠标和键盘输入。

函数

在默认环境中提供的许多值的类型为函数。函数是包裹在值中的程序片段。为了运行包裹的程序，可以将这些值应用于它们。例如，在浏览器环境中，绑定`prompt`包含一函数，个显示一个小对话框，请求用户输入。它是这样使用的：

```
prompt("Enter passcode");
```



执行一个函数被称为调用，或应用它（invoke，call，apply）。您可以通过在生成函数值的表达式之后放置括号来调用函数。通常你会直接使用持有该函数的绑定名称。括号之间的值被赋予函数内部的程序。在这个例子中，`prompt`函数使用我们提供的字符串作为文本来显示在对话框中。赋予函数的值称为参数。不同的函数可能需要不同的数量或不同类型的参数。

`prompt`函数在现代 Web 编程中用处不大，主要是因为你无法控制所得对话框的外观，但可以在玩具程序和实验中有所帮助。

`console.log` 函数

在例子中，我使用`console.log`来输出值。大多数 JavaScript 系统（包括所有现代 Web 浏览器和 Node.js）都提供了`console.log`函数，将其参数写入一个文本输出设备。在浏览器中，输出出现在 JavaScript 控制台中。浏览器界面的这一部分在默认情况下是隐藏的，但大多数浏览器在您按 F12 或在 Mac 上按 Command-Option-I 时打开它。如果这不起作用，请在菜单中搜索名为“开发人员工具”或类似的项目。

> 在英文版页面上运行示例（或自己的代码）时，会在示例之后显示`console.log`输出，而不是在浏览器的 JavaScript 控制台中显示。

```
let x = 30;
console.log("the value of x is", x);
// → the value of x is 30
```

尽管绑定名称不能包含句号字符，但是`console.log`确实拥有。这是因为`console.log`不是一个简单的绑定。它实际上是一个表达式，它从`console`绑定所持有的值中检索`log`属性。我们将在第 4 章中弄清楚这意味着什么。

返回值

显示对话框或将文字写入屏幕是一个副作用。由于它们产生的副作用，很多函数都很有用。函数也可能产生值，在这种情况下，他们不需要有副作用就有用。例如，函数`Math.max`可以接受任意数量的参数并返回最大值。

```
console.log(Math.max(2, 4));
// → 4
```

当一个函数产生一个值时，它被称为返回该值。任何产生值的东西都是 JavaScript 中的表达式，这意味着可以在较大的表达式中使用函数调用。在这里，`Math.min`的调用（与`Math.max`相反）用作加法表达式的一部分：

```
console.log(Math.min(2, 4) + 100);
// → 102
```

我们会在下一章当中讲解如何编写自定义函数。

控制流

当你的程序包含多个语句时，这些语句就像是一个故事一样从上到下执行。 这个示例程序有两个语句。 第一个要求用户输入一个数字，第二个在第一个之后执行，显示该数字的平方。

```
let theNumber = Number(prompt("Pick a number"));
console.log("Your number is the square root of " +
            theNumber * theNumber);
```

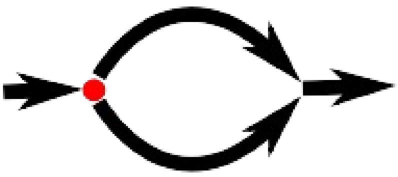
`Number` 函数将一个值转换为一个数字。 我们需要这种转换，因为 `prompt` 的结果是一个字符串值，我们需要一个数字。 有类似的函数叫做 `String` 和 `Boolean` ，它们将值转换为这些类型。

以下是直线控制流程的相当简单的示意图：



条件执行

并非所有的程序都是直路。 例如，我们可能想创建一条分叉路，在那里该程序根据当前的情况采取适当的分支。 这被称为条件执行。



在 JavaScript 中，条件执行使用 `if` 关键字创建。 在简单的情况下，当且仅当某些条件成立时，我们才希望执行一些代码。 例如，仅当输入实际上是一个数字时，我们可能打算显示输入的平方。

```
let theNumber = Number(prompt("Pick a number", ""));
if (!isNaN(theNumber))
  alert("Your number is the square root of " +
        theNumber * theNumber);
```

修改之后，如果您输入 `"parrot"` ，则不显示输出。

`if` 关键字根据布尔表达式的值执行或跳过语句。 决定性的表达式写在关键字之后，括号之间，然后是要执行的语句。

`Number.isNaN` 函数是一个标准的 JavaScript 函数，仅当它给出的参数是 `NaN` 时才返回 `true` 。当你给它一个不代表有效数字的字符串时，`Number` 函数恰好返回 `NaN` 。因此，条件翻译为“如果 `theNumber` 是一个数字，那么这样做”。

在这个例子中，`if` 下面的语句被大括号（`{` 和 `}`）括起来。它们可用于将任意数量的语句分组到单个语句中，称为代码块。在这种情况下，你也可以忽略它们，因为它们只包含一个语句，但为了避免必须考虑是否需要，大多数 JavaScript 程序员在每个这样的被包裹的语句中使用它们。除了偶尔的一行，我们在本书中大多会遵循这个约定。

```
if (1 + 1 == 2) console.log("It's true");
// → It's true
```

您通常不会只执行条件成立时代码，还会处理其他情况的代码。该替代路径由图中的第二个箭头表示。可以一起使用 `if` 和 `else` 关键字，创建两个单独的替代执行路径。

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
} else {
  console.log("Hey. Why didn't you give me a number?");
}
```

如果我们需要执行的路径多于两条，可以将多个 `if/else` 对链接在一起使用。如下所示例子：

```
let num = Number(prompt("Pick a number", "0"));

if (num < 10) {
  console.log("Small");
} else if (num < 100) {
  console.log("Medium");
} else {
  console.log("Large");
}
```

该程序首先会检查 `num` 是否小于 10。如果条件成立，则执行显示 `"Small"` 的这条路径；如果不成立，则选择 `else` 分支，`else` 分支自身包含了第二个 `if` 。如果第二个条件即 `num` 小于 100 成立，且数字的范围在 10 到 100 之间，则执行显示 `"Medium"` 的这条路径。如果上述条件均不满足，则执行最后一条 `else` 分支路径。

这个程序的模式看起来像这样：

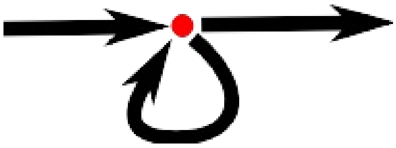


`while` 和 `do` 循环

现考虑编写一个程序，输出 0 到 12 之间的所有偶数。其中一种编写方式如下所示：

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

该程序确实可以工作，但编程的目的在于减少工作量，而非增加。如果我们需要小于 1000 的偶数，上面的方式是不可行的。我们现在所需的是重复执行某些代码的方法，我们将这种控制流程称为循环。



我们可以使用循环控制流来让程序执行回到之前的某个位置，并根据程序状态循环执行代码。如果我们在循环中使用一个绑定计数，那么就可以按照如下方式编写代码：

```
let number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
//   … etcetera
```

循环语句以关键字 `while` 开头。在关键字 `while` 后紧跟一个用括号括起来的表达式，括号后紧跟一条语句，这种形式与 `if` 语句类似。只要表达式产生的值转换为布尔值后为 `true`，该循环会持续进入括号后面的语句。

`number` 绑定演示了绑定可以跟踪程序进度的方式。 每次循环重复时， `number` 的值都比以前的值多 2。 在每次重复开始时，将其与数字 12 进行比较来决定程序的工作是否完成。

作为一个实际上有用的例子，现在我们可以编写一个程序来计算并显示 `2**10` （2 的 10 次方）的结果。 我们使用两个绑定：一个用于跟踪我们的结果，一个用来计算我们将这个结果乘以 2 的次数。 该循环测试第二个绑定是否已达到 10，如果不是，则更新这两个绑定。

```
let result = 1;
let counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

计数器也可以从 `1` 开始并检查 `<= 10` ，但是，由于一些在第 4 章中澄清的原因，从 0 开始计数是个好主意。

`do` 循环控制结构类似于 `while` 循环。两者之间只有一个区别： `do` 循环至少执行一遍循环体，只有第一次执行完循环体之后才会开始检测循环条件。 `do` 循环中将条件检测放在循环体后面，正反映了这一点：

```
let yourName;
do {
  yourName = prompt("Who are you?");
} while (!yourName);
console.log(yourName);
```

这个程序会强制你输入一个名字。 它会一再询问，直到它得到的东西不是空字符串。 `!` 运算符会将值转换为布尔类型再取反，除了 `""` 之外的所有字符串都转换为 `true` 。 这意味着循环持续进行，直到您提供了非空名称。

代码缩进

在这些例子中，我一直在语句前添加空格，它们是一些大型语句的一部分。 这些都不是必需的 - 没有它们，计算机也会接受该程序。 实际上，即使是程序中的换行符也是可选的。 如果你喜欢，你可以将程序编写为很长的一行。

块内缩进的作用是使代码结构显而易见。 在其他块内开启新的代码块中，可能很难看到块的结束位置，和另一个块开始位置。 通过适当的缩进，程序的视觉形状对应其内部块的形状。 我喜欢为每个开启的块使用两个空格，但风格不同 - 有些人使用四个空格，而有些人使用制表符。 重要的是，每个新块添加相同的空格量。

```
if (false != true) {
  console.log("That makes sense.");
  if (1 < 2) {
    console.log("No surprise there.");
  }
}
```

大多数代码编辑器程序（包括本书中的那个）将通过自动缩进新行来提供帮助。

`for` 循环

许多循环遵循 `while` 示例中看到的规律。 首先，创建一个计数器绑定来跟踪循环的进度。 然后出现一个 `while` 循环，通常用一个测试表达式来检查计数器是否已达到其最终值。 在循环体的末尾，更新计数器来跟踪进度。

由于这种规律非常常见，JavaScript 和类似的语言提供了一个稍短而且更全面的形式， `for` 循环：

```
for (let number = 0; number <= 12; number = number + 2)
  console.log(number);
// → 0
// → 2
//   … etcetera
```

该程序与之前的偶数打印示例完全等价。 唯一的变化是，所有与循环“状态”相关的语句，在`for`之后被组合在一起。

关键字`for`后面的括号中必须包含两个分号。第一个分号前面的是循环的初始化部分，通常是定义一个绑定。第二部分则是判断循环是否继续进行的检查表达式。最后一部分则是用于每个循环迭代后更新状态的语句。绝大多数情况下，`for`循环比`while`语句更简短清晰。

下面的代码中使用了`for`循环代替`while`循环，来计算`2**10`：

```
var result = 1;
for (var counter = 0; counter < 10; counter = counter + 1)
  result = result * 2;
console.log(result);
// → 1024
```

跳出循环

除了循环条件为`false`时循环会结束以外，我们还可以使用一个特殊的`break`语句来立即跳出循环。

下面的程序展示了`break`语句的用法。该程序的作用是找出第一个大于等于 20 且能被 7 整除的数字。

```
for (let current = 20; ; current++) {
  if (current % 7 == 0)
    break;
  }
}
// → 21
```

我们可以使用余数运算符（`%`）来判断一个数是否能被另一个数整除。如果可以整除，则余数为 0。

本例中的`for`语句省略了检查循环终止条件的表达式。这意味着除非执行了内部的`break`语句，否则循环永远不会结束。

如果你要删除这个`break`语句，或者你不小心写了一个总是产生`true`的结束条件，你的程序就会陷入死循环中。 死循环中的程序永远不会完成运行，这通常是一件坏事。

> 如果您在（英文版）这些页面的其中一个示例中创建了死限循环，则通常会询问您是否要在几秒钟后停止该脚本。 如果失败了，您将不得不关闭您正在处理的选项卡，或者在某些浏览器中关闭整个浏览器，以便恢复。

`continue` 关键字与 `break` 类似，也会对循环执行过程产生影响。循环体中的 `continue` 语句可以跳出循环体，并进入下一轮循环迭代。

更新绑定的简便方法

程序经常需要根据绑定的原值进行计算并更新值，特别是在循环过程中，这种情况更加常见。

```
counter = counter + 1;
```

JavaScript 提供了一种简便写法：

```
counter += 1;
```

JavaScript 还为其他运算符提供了类似的简便方法，比如 `result*=2` 可以将 `result` 变为原来的两倍，而 `counter-=1` 可以将 `counter` 减 1。

这样可以稍微简化我们的计数示例代码。

```
for (let number = 0; number <= 12; number += 2)
  console.log(number);
```

对于 `counter+=1` 和 `counter-=1`，还可以进一步简化代码，`counter+=1` 可以修改为 `counter++`，`counter-=1` 可以修改为 `counter--`。

`switch` 条件分支

我们很少会编写如下所示的代码。

```
if (x == "value1") action1();
else if (x == "value2") action2();
else if (x == "value3") action3();
else defaultAction();
```

有一种名为 `switch` 的结构，为了以更直接的方式表达这种“分发”。 不幸的是，JavaScript 为此所使用的语法（它从 C/Java 语言中继承而来）有些笨拙 - `if` 语句链看起来可能更好。 这里是一个例子：

```
switch (prompt("What is the weather like?")) {
  case "rainy":
    console.log("Remember to bring an umbrella.");
    break;
  case "sunny":
    console.log("Dress lightly.");
  case "cloudy":
    console.log("Go outside.");
    break;
  default:
    console.log("Unknown weather type!");
    break;
}
```

你可以在 `switch` 打开的块内放置任意数量的 `case` 标签。 程序会在向 `switch` 提供的值的对应标签处开始执行，或者如果没有找到匹配值，则在 `default` 处开始。 甚至跨越了其他标签，它也会继续执行，直到达到了 `break` 声明。 在某些情况下，例如在示例中的 `"sunny"` 的情况下，这可以用来在不同情况下共享一些代码（它建议在晴天和多云天气外出）。 但要小心 - 很容易忘记这样的 `break` ，这会导致程序执行你不想执行的代码。

大写

绑定名中不能包含空格，但很多时候使用多个单词有助于清晰表达绑定的实际用途。当绑定名中包含多个单词时可以选择多种写法，以下是可以选择的几种绑定名书写方式：

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

第一种风格可能很难阅读。 我更喜欢下划线的外观，尽管这种风格有点痛苦。 标准的 JavaScript 函数和大多数 JavaScript 程序员都遵循最底下的风格 - 除了第一个词以外，它们都会将每个词的首字母大写。 要习惯这样的小事并不困难，而且混合命名风格的代码可能会让人反感，所以我们遵循这个约定。

在极少数情况下，绑定名首字母也会大写，比如 `Number` 函数。这种方式用来表示该函数是构造函数。我们会在第6章详细讲解构造函数的概念。现在，我们没有必要纠结于表面上的风格不一致性。

注释

通常，原始代码并不能传达你让一个程序传达给读者的所有信息，或者它以神秘的方式传达信息，人们可能不了解它。 在其他时候，你可能只想包含一些相关的想法，作为你程序的一部分。 这是注释的用途。

注释是程序中的一段文本，而在程序执行时计算机会完全忽略掉这些文本。JavaScript 中编写注释有两种方法，写单行注释时，使用两个斜杠字符开头，并在后面添加文本注释。

```
let accountBalance = calculateBalance(account);
// It's a green hollow where a river sings
accountBalance.adjust();
// Madly catching white tatters in the grass.
let report = new Report();
// Where the sun on the proud mountain rings:
addToReport(accountBalance, report);
// It's a little valley, foaming like light in a glass.
```

`//` 注释只能到达行尾。 `/*` 和 `*/` 之间的一段文本将被忽略，不管它是否包含换行符。 这对添加文件或程序块的信息块很有用。

```
/*
 I first found this number scrawled on the back of one of
 an old notebook. Since then, it has often dropped by,
 showing up in phone numbers and the serial numbers of
 products that I've bought. It obviously likes me, so I've
 decided to keep it.
*/
const myNumber = 11213;
```

本章小结

在本章中，我们学习并了解了程序由语句组成，而每条语句又有可能包含了更多语句。在语句中往往包含了表达式，而表达式还可以由更小的表达式组成。

程序中的语句按顺序编写，并从上到下执行。你可以使用条件语句（`if`、`else`和`switch`）或循环语句（`while`、`do`和`for`）来改变程序的控制流。

绑定可以用来保存任何数据，并用一个绑定名对其引用。而且在记录你的程序执行状态时十分有用。环境是一组定义好的绑定集合。JavaScript 的运行环境中总会包含一系列有用的标准绑定。

函数是一种特殊的值，用于封装一段程序。你可以通过`functionName(arg1, arg2)`这种写法来调用函数。函数调用可以是一个表达式，也可以用于生成一个值。

习题

如果你不清楚在哪里可以找到习题的提示，请参考本书的简介部分。

每个练习都以问题描述开始。 阅读并尝试解决这个练习。 如果遇到问题，请考虑阅读练习后的提示。 本书不包含练习的完整解决方案，但您可以在 eloquentjavascript.net/code 上在线查找它们。 如果你想从练习中学到一些东西，我建议仅在你解决了这个练习之后，或者至少在你努力了很长时间而感到头疼之后，再看看这些解决方案。

LoopingaTriangle

编写一个循环，调用 7 次`console.log`函数，打印出如下的三角形：

```
#
##
##
###
###
####
#####
```

这里给出一个小技巧，在字符串后加上`.length`可以获取字符串的长度。

```
let abc = "abc";
console.log(abc.length);
// → 3
```

FizzBuzz

编写一个程序，使用`console.log`打印出从 1 到 100 的所有数字。不过有两种例外情况：当数字能被 3 整除时，不打印数字，而打印`"Fizz"`。当数字能被 5 整除时（但不能被 3 整除），不打印数字，而打印`"Buzz"`。

当以上程序可以正确运行后，请修改你的程序，让程序在遇到能同时被 3 与 5 整除的数字时，打印出`"FizzBuzz"`。

（这实际上是一个面试问题，据说剔除了很大一部分程序员候选人，所以如果你解决了这个问题，你的劳动力市场价值就会上升。）

棋盘

编写一个程序，创建一个字符串，用于表示`8×8`的网格，并使用换行符分隔行。网格中的每个位置可以是空格或字符`"#"`。这些字符组成了一张棋盘。

将字符串传递给`console.log`将会输出以下结果：

```
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
```

当程序可以产生这样的输出后，请定义绑定`size=8`，并修改程序，使程序可以处理任意尺寸（长宽由`size`确定）的棋盘，并输出给定宽度和高度的网格。

三、函数

> 原文：Functions
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 人们认为计算机科学是天才的艺术，但是实际情况相反，只是许多人在其它人基础上做一些东西，就像一面由石子垒成的墙。
>
> 高德纳



函数是 JavaScript 编程的面包和黄油。 将一段程序包装成值的概念有很多用途。 它为我们提供了方法，用于构建更大程序，减少重复，将名称和子程序关联，以及将这些子程序相互隔离。

函数最明显的应用是定义新词汇。 用散文创造新词汇通常是不好的风格。 但在编程中，它是不可或缺的。

以英语为母语的典型成年人，大约有 2 万字的词汇量。 很少有编程语言内置了 2 万个命令。 而且，可用的词汇的定义往往比人类语言更精确，因此灵活性更低。 因此，我们通常会引入新的概念，来避免过多重复。

定义函数

函数定义是一个常规绑定，其中绑定的值是一个函数。 例如，这段代码定义了 `square`，来引用一个函数，它产生给定数字的平方：

```
const square = function(x) {
  return x * x;
};

console.log(square(12));
// → 144
```

函数使用以关键字 `function` 起始的表达式创建。 函数有一组参数（在本例中只有 `x`）和一个主体，它包含调用该函数时要执行的语句。 以这种方式创建的函数的函数体，必须始终包在花括号中，即使它仅包含一个语句。

一个函数可以包含多个参数，也可以不含参数。在下面的例子中，`makeNoise` 函数中没有包含任何参数，而 `power` 则使用了两个参数：

```
var makeNoise = function() {
  console.log("Pling!");
};

makeNoise();
// → Pling!

const power = function(base, exponent) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
     result *= base;
  }
  return result;
};

console.log(power(2, 10));
// → 1024
```

有些函数会产生一个值，比如 `power` 和 `square`，有些函数不会，比如 `makeNoise`，它的唯一结果是副作用。 `return` 语句决定函数返回的值。 当控制流遇到这样的语句时，它立即跳出当前函数并将返回的值赋给调用该函数的代码。 不带表达式的 `return` 关键字，会导致函数返回 `undefined`。 没有 `return` 语句的函数，比如 `makeNoise`，同样返回 `undefined`。

函数的参数行为与常规绑定相似，但它们的初始值由函数的调用者提供，而不是函数本身的代码。

绑定和作用域

每个绑定都有一个作用域，它是程序的一部分，其中绑定是可见的。 对于在任何函数或块之外定义的绑定，作用域是整个程序 - 您可以在任何地方引用这种绑定。它们被称为全局的。

但是为函数参数创建的，或在函数内部声明的绑定，只能在该函数中引用，所以它们被称为局部绑定。 每次调用该函数时，都会创建这些绑定的新实例。 这提供了函数之间的一些隔离 - 每个函数调用，都在它自己的小世界（它的局部环境）中运行，并且通常可以在不知道全局

环境中发生的事情的情况下理解。

用 `let` 和 `const` 声明的绑定，实际上是它们的声明所在的块的局部对象，所以如果你在循环中创建了一个，那么循环之前和之后的代码就不能“看见”它。JavaScript 2015 之前，只有函数创建新的作用域，因此，使用 `var` 关键字创建的旧式绑定，在它们出现的整个函数中内都可见，或者如果它们不在函数中，在全局作用域可见。

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);
  // → 60
}
// y is not visible here
console.log(x + z);
// → 40
```

每个作用域都可以“向外查看”它周围的作用域，所以示例中的块内可以看到 `x` 。 当多个绑定具有相同名称时例外 - 在这种情况下，代码只能看到最内层的那个。 例如，当 `halve` 函数中的代码引用 `n` 时，它看到它自己的 `n` ，而不是全局的 `n` 。

```
const halve = function(n) {
  return n / 2;
}
let n = 10;
console.log(halve(100));
// → 50
console.log(n);
// → 10
```

嵌套作用域

JavaScript 不仅区分全局和局部绑定。 块和函数可以在其他块和函数内部创建，产生多层局部环境。

例如，这个函数（输出制作一批鹰嘴豆泥所需的配料）的内部有另一个函数：

```
const hummus = function(factor) {
  const ingredient = function(amount, unit, name) {
    let ingredientAmount = amount * factor;
    if (ingredientAmount > 1) {
      unit += "s";
    }
    console.log(`${ingredientAmount} ${unit} ${name}`);
  };
  ingredient(1, "can", "chickpeas");
  ingredient(0.25, "cup", "tahini");
  ingredient(0.25, "cup", "lemon juice");
  ingredient(1, "clove", "garlic");
  ingredient(2, "tablespoon", "olive oil");
  ingredient(0.5, "teaspoon", "cumin");
};
```

`ingredient`函数中的代码，可以从外部函数中看到`factor`绑定。 但是它的局部绑定，比如`unit`或`ingredientAmount`，在外层函数中是不可见的。

简而言之，每个局部作用域也可以看到所有包含它的局部作用域。 块内可见的绑定集，由这个块在程序文本中的位置决定。 每个局部作用域也可以看到包含它的所有局部作用域，并且所有作用域都可以看到全局作用域。 这种绑定可见性方法称为词法作用域。

作为值的函数

函数绑定通常只充当程序特定部分的名称。 这样的绑定被定义一次，永远不会改变。 这使得容易混淆函数和名称。

```
let launchMissiles = function(value) {
  missileSystem.launch("now");
};
if (safeMode) {
  launchMissiles = function() {/* do nothing */};
}
```

在第 5 章中，我们将会讨论一些高级功能：将函数类型的值传递给其他函数。

符号声明

创建函数绑定的方法稍短。 当在语句开头使用`function`关键字时，它的工作方式不同。

```
function square(x) {
  return x * x;
}
```

这是函数声明。 该语句定义了绑定`square`并将其指向给定的函数。 写起来稍微容易一些，并且在函数之后不需要分号。

这种形式的函数定义有一个微妙之处。

```
console.log("The future says:", future());

function future() {
  return "You'll never have flying cars";
}
```

前面的代码可以执行，即使在函数定义在使用它的代码下面。 函数声明不是常规的从上到下的控制流的一部分。 在概念上，它们移到了其作用域的顶部，并可被该作用域内的所有代码使用。 这有时是有用的，因为它以一种看似有意义的方式，提供了对代码进行排序的自由，而无需担心在使用之前必须定义所有函数。

箭头函数

函数的第三个符号与其他函数看起来有很大不同。 它不使用 `function` 关键字，而是使用由等号和大于号组成的箭头（ `=>` ）（不要与大于等于运算符混淆，该运算符写做 `>=` ）。

```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

箭头出现在参数列表后面，然后是函数的主体。 它表达了一些东西，类似“这个输入（参数）产生这个结果（主体）”。

如果只有一个参数名称，则可以省略参数列表周围的括号。 如果主体是单个表达式，而不是大括号中的块，则表达式将从函数返回。 所以这两个 `square` 的定义是一样的：

```
const square1 = (x) => { return x * x; };
const square2 = x => x * x;
```

当一个箭头函数没有参数时，它的参数列表只是一组空括号。

```
const horn = () => {
  console.log("Toot");
};
```

在语言中没有很好的理由，同时拥有箭头函数和函数表达式。 除了我们将在第 6 章中讨论的一个小细节外，他们实现相同的东西。 在 2015 年增加了箭头函数，主要是为了能够以简短的方式编写小函数表达式。 我们将在第 5 章中使用它们。

调用栈

控制流经过函数的方式有点复杂。 让我们仔细看看它。 这是一个简单的程序，它执行了一些函数调用：

```
function greet(who) {
  console.log("Hello " + who);
}
greet("Harry");
console.log("Bye");
```

这个程序的执行大致是这样的：对 `greet` 的调用使控制流跳转到该函数的开始（第 2 行）。该函数调用控制台的 `console.log` 来完成它的工作，然后将控制流返回到第 2 行。 它到达 `greet` 函数的末尾，所以它返回到调用它的地方，这是第 4 行。 之后的一行再次调

用 `console.log` 。之后，程序结束。

我们可以使用下图表示出控制流：

```
not in function
   in greet
        in console.log
   in greet
not in function
   in console.log
not in function
```

由于函数在返回时必须跳回调用它的地方，因此计算机必须记住调用发生处上下文。 在一种情况下， `console.log` 完成后必须返回 `greet` 函数。 在另一种情况下，它返回到程序的结尾。

计算机存储此上下文的地方是调用栈。 每次调用函数时，当前上下文都存储在此栈的顶部。当函数返回时，它会从栈中删除顶部上下文，并使用该上下文继续执行。

存储这个栈需要计算机内存中的空间。 当栈变得太大时，计算机将失败，并显示“栈空间不足”或“递归太多”等消息。 下面的代码通过向计算机提出一个非常困难的问题来说明这一点，这个问题会导致两个函数之间的无限的来回调用。 相反，如果计算机有无限的栈，它将会是无限的。 事实上，我们将耗尽空间，或者“把栈顶破”。

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
console.log(chicken() + " came first.");
// → ??
```

可选参数

下面的代码可以正常执行：

```
function square(x) { return x * x; }
console.log(square(4, true, "hedgehog"));
// → 16
```

我们定义了 `square` ，只带有一个参数。 然而，当我们使用三个参数调用它时，语言并不会报错。 它会忽略额外的参数并计算第一个参数的平方。

JavaScript 对传入函数的参数数量几乎不做任何限制。如果你传递了过多参数，多余的参数就会被忽略掉，而如果你传递的参数过少，遗漏的参数将会被赋值成 `undefined` 。

该特性的缺点是你可能恰好向函数传递了错误数量的参数，但没有人会告诉你这个错误。

优点是这种行为可以用于使用不同数量的参数调用一个函数。 例如，这个`minus`函数试图通过作用于一个或两个参数，来模仿`-`运算符：

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

如果你在一个参数后面写了一个`=`运算符，然后是一个表达式，那么当没有提供它时，该表达式的值将会替换该参数。

例如，这个版本的`power`使其第二个参数是可选的。 如果你没有提供或传递`undefined`，它将默认为 2，函数的行为就像`square`。

```
function power(base, exponent = 2) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
}

console.log(power(4));
// → 16
console.log(power(2, 6));
// → 64
```

在下一章当中，我们将会了解如何获取传递给函数的整个参数列表。我们可以借助于这种特性来实现函数接收任意数量的参数。比如`console.log`就利用了这种特性，它可以用来输出所有传递给它的值。

```
console.log("C", "O", 2);
// → C O 2
```

闭包

函数可以作为值使用，而且其局部绑定会在每次函数调用时重新创建，由此引出一个值得我们探讨的问题：如果函数已经执行结束，那么这些由函数创建的局部绑定会如何处理呢？

下面的示例代码展示了这种情况。代码中定义了函数`wrapValue`，该函数创建了一个局部绑定`localVariable`，并返回一个函数，用于访问并返回局部绑定`localVariable`。

```
function wrapValue(n) {
  let local = n;
  return () => local;
}

let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

这是允许的并且按照您的希望运行 - 绑定的两个实例仍然可以访问。 这种情况很好地证明了一个事实，每次调用都会重新创建局部绑定，而且不同的调用不能覆盖彼此的局部绑定。

这种特性（可以引用封闭作用域中的局部绑定的特定实例）称为闭包。 引用来自周围的局部作用域的绑定的函数称为（一个）闭包。 这种行为不仅可以让您免于担心绑定的生命周期，而且还可以以创造性的方式使用函数值。

我们对上面那个例子稍加修改，就可以创建一个可以乘以任意数字的函数。

```
function multiplier(factor) {
 return number => number * factor;
}

let twice = multiplier(2);
console.log(twice(5));
// → 10
```

由于参数本身就是一个局部绑定，所以 `wrapValue` 示例中显式的 `local` 绑定并不是真的需要。

考虑这样的程序需要一些实践。 一个好的心智模型是，将函数值看作值，包含他们主体中的代码和它们的创建环境。 被调用时，函数体会看到它的创建环境，而不是它的调用环境。

这个例子调用 `multiplier` 并创建一个环境，其中 `factor` 参数绑定了 2。 它返回的函数值，存储在 `twice` 中，会记住这个环境。 所以当它被调用时，它将它的参数乘以 2。

递归

一个函数调用自己是完全可以的，只要它没有经常这样做以致溢出栈。 调用自己的函数被称为递归函数。 递归允许一些函数以不同的风格编写。 举个例子，这是 `power` 的替代实现：

```
function power(base, exponent) {
  if (exponent == 0) {
    return 1;
  } else {
    return base * power(base, exponent - 1);
  }
}

console.log(power(2, 3));
// → 8
```

这与数学家定义幂运算的方式非常接近，并且可以比循环变体将该概念描述得更清楚。 该函数以更小的指数多次调用自己以实现重复的乘法。

但是这个实现有一个问题：在典型的 JavaScript 实现中，它大约比循环版本慢三倍。 通过简单循环来运行，通常比多次调用函数开销低。

速度与优雅的困境是一个有趣的问题。 您可以将其视为人性化和机器友好性之间的权衡。 几乎所有的程序都可以通过更大更复杂的方式加速。 程序员必须达到适当的平衡。

在 `power` 函数的情况下，不雅的（循环）版本仍然非常简单易读。 用递归版本替换它没有什么意义。 然而，通常情况下，一个程序处理相当复杂的概念，为了让程序更直接，放弃一些效率是有帮助的。

担心效率可能会令人分心。 这又是另一个让程序设计变复杂的因素，当你做了一件已经很困难的事情时，担心的额外事情可能会瘫痪。

因此，总是先写一些正确且容易理解的东西。 如果您担心速度太慢 - 通常不是这样，因为大多数代码的执行不足以花费大量时间 - 您可以事后进行测量并在必要时进行改进。

递归并不总是循环的低效率替代方法。 递归比循环更容易解决解决一些问题。 这些问题通常是需要探索或处理几个“分支”的问题，每个“分支”可能再次派生为更多的分支。

考虑这个难题：从数字 1 开始，反复加 5 或乘 3，就可以产生无限数量的新数字。 你会如何编写一个函数，给定一个数字，它试图找出产生这个数字的，这种加法和乘法的序列？

例如，数字 13 可以通过先乘 3 然后再加 5 两次来到达，而数字 15 根本无法到达。

使用递归编码的解决方案如下所示：

```
function findSolution(target) {
  function find(current, history) {
    if (current == target) {
      return history;
    } else if (current > target) {
      return null;
    } else {
      return find(current + 5, `(${history} + 5)`) ||
             find(current * 3, `(${history} * 3)`);
    }
  }
  return find(1, "1");
}

console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)
```

需要注意的是该程序并不需要找出最短运算序列，只需要找出任何一个满足要求的序列即可。

如果你没有看到它的工作原理，那也没关系。让我们浏览它，因为它是递归思维的很好的练习。

内层函数 `find` 进行实际的递归。它有两个参数：当前数字和记录我们如何到达这个数字的字符串。如果找到解决方案，它会返回一个字符串，显示如何到达目标。如果从这个数字开始找不到解决方案，则返回 `null`。

为此，该函数执行三个操作之一。如果当前数字是目标数字，则当前历史记录是到达目标的一种方式，因此将其返回。如果当前的数字大于目标，则进一步探索该分支是没有意义的，因为加法和乘法只会使数字变大，所以它返回 `null`。最后，如果我们仍然低于目标数字，函数会尝试从当前数字开始的两个可能路径，通过调用它自己两次，一次是加法，一次是乘法。如果第一次调用返回非 `null` 的东西，则返回它。否则，返回第二个调用，无论它产生字符串还是 `null`。

为了更好地理解函数执行过程，让我们来看一下搜索数字 13 时，`find` 函数的调用情况：

```
find(1, "1")
  find(6, "(1 + 5)")
    find(11, "((1 + 5) + 5)")
      find(16, "(((1 + 5) + 5) + 5)")
        too big
      find(33, "(((1 + 5) + 5) * 3)")
        too big
    find(18, "((1 + 5) * 3)")
      too big
  find(3, "(1 * 3)")
    find(8, "((1 * 3) + 5)")
      find(13, "(((1 * 3) + 5) + 5)")
        found!
```

缩进表示调用栈的深度。第一次调用 `find` 时，它首先调用自己来探索以 `(1 + 5)` 开始的解决方案。这一调用将进一步递归，来探索每个后续的解，它产生小于或等于目标数字。由于它没有找到一个命中目标的解，所以它向第一个调用返回 `null`。那里的 `||` 操作符会使探

索 `(1 * 3)` 的调用发生。 这个搜索的运气更好 - 它的第一次递归调用，通过另一个递归调用，命中了目标数字。 最内层的调用返回一个字符串，并且中间调用中的每个“||”运算符都会传递该字符串，最终返回解决方案。

添加新函数

这里有两种常用的方法，将函数引入到程序中。

首先是你发现自己写了很多次非常相似的代码。 我们最好不要这样做。 拥有更多的代码，意味着更多的错误空间，并且想要了解程序的人阅读更多资料。 所以我们选取重复的功能，为它找到一个好名字，并把它放到一个函数中。

第二种方法是，你发现你需要一些你还没有写的功能，这听起来像是它应该有自己的函数。 您将首先命名该函数，然后您将编写它的主体。 在实际定义函数本身之前，您甚至可能会开始编写使用该函数的代码。

给函数起名的难易程度取决于我们封装的函数的用途是否明确。对此，我们一起来看一个例子。

我们想编写一个打印两个数字的程序，第一个数字是农场中牛的数量，第二个数字是农场中鸡的数量，并在数字后面跟上 `Cows` 和 `Chickens` 用以说明，并且在两个数字前填充 0，以使得每个数字总是由三位数字组成。

```
007 Cows
011 Chickens
```

这需要两个参数的函数 - 牛的数量和鸡的数量。 让我们来编程。

```
function printFarmInventory(cows, chickens) {
  let cowString = String(cows);
  while (cowString.length < 3) {
    cowString = "0" + cowString;
  }
  console.log(`${cowString} Cows`);
  let chickenString = String(chickens);
  while (chickenString.length < 3) {
    chickenString = "0" + chickenString;
  }
  console.log(`${chickenString} Chickens`);
}
printFarmInventory(7, 11);
```

在字符串表达式后面写 `.length` 会给我们这个字符串的长度。 因此， `while` 循环在数字字符串前面加上零，直到它们至少有三个字符的长度。

任务完成！ 但就在我们即将向农民发送代码（连同大量发票）时，她打电话告诉我们，她也开始饲养猪，我们是否可以扩展软件来打印猪的数量？

当然没有问题。但是当再次复制粘贴这四行代码的时候，我们停了下来并重新思考。一定还有更好的方案来解决我们的问题。以下是第一种尝试：

```
function printZeroPaddedWithLabel(number, label) {
  let numberString = String(number);
  while (numberString.length < 3) {
    numberString = "0" + numberString;
  }
  console.log(`${numberString} ${label}`);
}

function printFarmInventory(cows, chickens, pigs) {
  printZeroPaddedWithLabel(cows, "Cows");
  printZeroPaddedWithLabel(chickens, "Chickens");
  printZeroPaddedWithLabel(pigs, "Pigs");
}

printFarmInventory(7, 11, 3);
```

这种方法解决了我们的问题！但是`printZeroPaddedWithLabel`这个函数并不十分恰当。它把三个操作，即打印信息、数字补零和添加标签放到了一个函数中处理。

这一次，我们不再将程序当中重复的代码提取成一个函数，而只是提取其中一项操作。

```
function zeroPad(number, width) {
  let string = String(number);
  while (string.length < width) {
    string = "0" + string;
  }
  return string;
}

function printFarmInventory(cows, chickens, pigs) {
  console.log(`${zeroPad(cows, 3)} Cows`);
  console.log(`${zeroPad(chickens, 3)} Chickens`);
  console.log(`${zeroPad(pigs, 3)} Pigs`);
}

printFarmInventory(7, 16, 3);
```

名为`zeroPad`的函数具有很好的名称，使读取代码的人更容易弄清它的功能。 而且这样的函数在更多的情况下是有用的，不仅仅是这个特定程序。 例如，您可以使用它来帮助打印精确对齐的数字表格。

我们的函数应该包括多少功能呢？我们可以编写一个非常简单的函数，只支持将数字扩展成 3 字符宽。也可以编写一个复杂通用的数字格式化系统，可以处理分数、负数、小数点对齐和使用不同字符填充等。

一个实用原则是不要故作聪明，除非你确定你会需要它。 为你遇到的每一个功能编写通用“框架”是很诱人的。 控制住那种冲动。 你不会完成任何真正的工作 - 你只会编写你永远不会使用的代码。

函数及其副作用

我们可以将函数分成两类：一类调用后产生副作用，而另一类则产生返回值（当然我们也可以定义同时产生副作用和返回值的函数）。

在农场案例当中，我们调用第一个辅助函数 `printZeroPaddedWithLabel` 来产生副作用，打印一行文本信息。而在第二个版本中有一个 `zeroPad` 函数，我们调用它来产生返回值。第二个函数比第一个函数的应用场景更加广泛，这并非偶然。相比于直接产生副作用的函数，产生返回值的函数则更容易集成到新的环境当中使用。

纯函数是一种特定类型的，生成值的函数，它不仅没有副作用，而且也不依赖其他代码的副作用，例如，它不读取值可能会改变的全局绑定。 纯函数具有令人愉快的属性，当用相同的参数调用它时，它总是产生相同的值（并且不会做任何其他操作）。 这种函数的调用，可以由它的返回值代替而不改变代码的含义。 当你不确定纯函数是否正常工作时，你可以通过简单地调用它来测试它，并且知道如果它在当前上下文中工作，它将在任何上下文中工作。 非纯函数往往需要更多的脚手架来测试。

尽管如此，我们也没有必要觉得非纯函数就不好，然后将这类函数从代码中删除。副作用常常是非常有用的。比如说，我们不可能去编写一个纯函数版本的 `console.log`，但 `console.log` 依然十分实用。而在副作用的帮助下，有些操作则更易、更快实现，因此考虑到运算速度，有时候纯函数并不可取。

本章小结

本章教你如何编写自己的函数。 当用作表达式时，`function` 关键字可以创建一个函数值。 当作为一个语句使用时，它可以用来声明一个绑定，并给它一个函数作为它的值。 箭头函数是另一种创建函数的方式。

```
// Define f to hold a function value
const f = function(a) {
  console.log(a + 2);
};

// Declare g to be a function
function g(a, b) {
  return a * b * 3.5;
}

// A less verbose function value
let h = a => a % 3;
```

理解函数的一个关键方面是理解作用域。 每个块创建一个新的作用域。 在给定作用域内声明的参数和绑定是局部的，并且从外部看不到。 用 `var` 声明的绑定行为不同 - 它们最终在最近的函数作用域或全局作用域内。

将程序执行的任务分成不同的功能是有帮助的。 你不必重复自己，函数可以通过将代码分组成一些具体事物，来组织程序。

习题

最小值

前一章介绍了标准函数`Math.min`，它可以返回参数中的最小值。我们现在可以构建相似的东西。编写一个函数`min`，接受两个参数，并返回其最小值。

```
// Your code here.

console.log(min(0, 10));
// → 0
console.log(min(0, -10));
// → -10
```

递归

我们已经看到，`%`（取余运算符）可以用于判断一个数是否是偶数，通过使用`% 2`来检查它是否被 2 整除。这里有另一种方法来判断一个数字是偶数还是奇数：

- 0是偶数
- 1是奇数
- 对于其他任何数字N，其奇偶性与N–2相同。

定义对应此描述的递归函数`isEven`。 该函数应该接受一个参数（一个正整数）并返回一个布尔值。

使用 50 与 75 测试该函数。想想如果参数为 –1 会发生什么以及产生相应结果的原因。请你想一个方法来修正该问题。

```
// Your code here.

console.log(isEven(50));
// → true
console.log(isEven(75));
// → false
console.log(isEven(-1));
// → ??
```

字符计数

你可以通过编写`"string"[N]`，来从字符串中得到第`N`个字符或字母。 返回的值将是只包含一个字符的字符串（例如`"b"`）。 第一个字符的位置为零，这会使最后一个字符在`string.length - 1`。 换句话说，含有两个字符的字符串的长度为2，其字符的位置为 0 和 1。

编写一个函数`countBs`，接受一个字符串参数，并返回一个数字，表示该字符串中有多少个大写字母`"B"`。

接着编写一个函数`countChar`，和`countBs`作用一样，唯一区别是接受第二个参数，指定需要统计的字符（而不仅仅能统计大写字母`"B"`）。并使用这个新函数重写函数`countBs`。

```
// Your code here.

console.log(countBs("BBC"));
// → 2
console.log(countChar("kakkerlak", "k"));
// → 4
```

四、数据结构：对象和数组

> 原文：Data Structures: Objects and Arrays
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> On two occasions I have been asked, ‘Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?’ [...] I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
>
> Charles Babbage，《Passages from the Life of a Philosopher》（1864）



数字，布尔和字符串是构建数据结构的原子。 不过，许多类型的信息都需要多个原子。 对象允许我们将值（包括其他对象）放到一起，来构建更复杂的结构。

我们迄今为止构建的程序，受到一个事实的限制，它们仅在简单数据类型上运行。 本章将介绍基本的数据结构。 到最后，你会知道足够多的东西，开始编写有用的程序。

本章将着手于一个或多或少的实际编程示例，当概念适用于手头问题时引入它们。 示例代码通常基于本文前面介绍的函数和绑定。

松鼠人

一般在晚上八点到十点之间，雅克就会变身成为一只毛茸茸的松鼠，尾巴上的毛十分浓密。

一方面，雅克非常高兴他没有变成经典的狼人。 与变成狼相比，变成松鼠的确会产生更少的问题。 他不必担心偶然吃掉邻居（那会很尴尬），而是担心被邻居的猫吃掉。 他在橡木树冠上的一个薄薄的树枝上醒来，赤身裸体并迷失方向。在这两次偶然之后，他在晚上锁上了房间的门窗，并在地板上放了几个核桃，来使自己忙起来。

这就解决了猫和树的问题。 但雅克宁愿完全摆脱他的状况。 不规律发生的变身使他怀疑，它们可能会由某种东西触发。 有一段时间，他相信只有在他靠近橡树的日子里才会发生。 但是避开橡树不能阻止这个问题。

雅克切换到了更科学的方法，开始每天记录他在某一天所做的每件事，以及他是否变身。 有了这些数据，他希望能够缩小触发变身的条件。

他需要的第一个东西，是存储这些信息的数据结构。

数据集

为了处理大量的数字数据，我们首先必须找到一种方法，将其在我们的机器内存中表示。 举例来说，我们想要表示一组数字 2, 3, 5, 7 和 11。

我们可以用字符串来创建 - 毕竟，字符串可以有任意长度，所以我们可以把大量数据放入它们中，并使用`"2 3 5 7 11"`作为我们的表示。 但这很笨拙。 你必须以某种方式提取数字，并将它们转换回数字才能访问它们。

幸运的是，JavaScript提供了一种数据类型，专门用于存储一系列的值。我们将这种数据类型称为数组，将一连串的值写在方括号当中，值之间使用逗号（`,`）分隔。

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);
// → 5
console.log(listOfNumbers[0]);
// → 2
console.log(listOfNumbers[2 - 1]);
// → 3
```

我们同样使用方括号来获取数组当中的值。在表达式后紧跟一对方括号，并在方括号中填写表达式，这将会在左侧表达式里查找方括号中给定的索引所对应的值，并返回结果。

数组的第一个索引是零，而不是一。 所以第一个元素用`listOfNumbers[0]`获取。 基于零的计数在技术上有着悠久的传统，并且在某些方面意义很大，但需要一些时间来习惯。 将索引看作要跳过的项目数量，从数组的开头计数。

属性

在之前的章节中，我们已经看到了一些可疑的表达式，例如 `myString.length`（获取字符串的长度）和 `Math.max`（最大值函数）。这些表达式可以访问某个值的属性。在第一个中，我们访问 `myString` 中的 `length` 属性。第二个中，我们访问 `Math` 对象（它是数学相关常量和函数的集合）中的名为 `max` 的属性。

在 JavaScript 中，几乎所有的值都有属性。但 `null` 和 `undefined` 没有。如果你尝试访问 `null` 和 `undefined` 的属性，会得到一个错误提示。

```
null.length;
// → TypeError: null has no properties
```

在JavaScript中访问属性的两种主要方式是点（`.`）和方括号（`[]`）。`value.x` 和 `value [x]` 都可以访问 `value` 属性，但不一定是同一个属性。区别在于如何解释 `x`。使用点时，点后面的单词是该属性的字面名称。使用方括号时，会求解括号内的表达式来获取属性名称。鉴于 `value.x` 获取 `value` 的名为 `x` 的属性，`value [x]` 尝试求解表达式 `x`，并将结果转换为字符串作为属性名称。

所以如果你知道你感兴趣的属性叫做 `color`，那么你会写 `value.color`。如果你想提取属性由绑定 `i` 中保存的值命名，你可以写 `value [i]`。属性名称是字符串。它们可以是任何字符串，但点符号仅适用于看起来像有效绑定名的名称。所以如果你想访问名为 `2` 或 `John Doe` 的属性，你必须使用方括号：`value[2]` 或 `value["John Doe"]`。

数组中的元素以数组属性的形式存储，使用数字作为属性名称。因为你不能用点号来表示数字，并且通常想要使用一个保存索引的绑定，所以你必须使用括号来表达它们。

数组的 `length` 属性告诉我们它有多少个元素。这个属性名是一个有效的绑定名，我们事先知道它的名字，所以为了得到一个数组的长度，通常写 `array.length`，因为它比 `array["length"]` 更容易编写。

方法

了 `length` 属性之外，字符串和数组对象都包含一些持有函数值的属性。

```
let doh = "Doh";
console.log(typeof doh.toUpperCase);
// → function
console.log(doh.toUpperCase());
// → DOH
```

每个字符串都有 `toUpperCase` 属性。调用时，它将返回所有字母转换为大写字符串的副本。另外还有 `toLowerCase`。

有趣的是，虽然我们没有在调用 `toUpperCase` 时传递任何参数，但该函数访问了字符串 `"Doh"`，即被调用的属性所属的值。我们会在第 6 章中阐述这其中的原理。

我们通常将包含函数的属性称为某个值的方法。比如说，`toUpperCase`是字符串的一个方法。

此示例演示了两种方法，可用于操作数组：

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence);
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

`push`方法将值添加到数组的末尾，而`pop`方法则相反，删除数组中的最后一个值并将其返回。

这些有点愚蠢的名字是栈的传统术语。 编程中的栈是一种数据结构，它允许你将值推入并按相反顺序再次弹出，最后添加的内容首先被移除。 这些在编程中很常见 - 你可能还记得前一章中的函数调用栈，它是同一个想法的实例。

对象

回到松鼠人的示例。 一组每日的日志条目可以表示为一个数组。 但是这些条目并不仅仅由一个数字或一个字符串组成 - 每个条目需要存储一系列活动和一个布尔值，表明雅克是否变成了松鼠。 理想情况下，我们希望将它们组合成一个值，然后将这些分组的值放入日志条目的数组中。

对象类型的值是任意的属性集合。 创建对象的一种方法是使用大括号作为表达式。

```
let day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

大括号内有一列用逗号分隔的属性。 每个属性都有一个名字，后跟一个冒号和一个值。 当一个对象写为多行时，像这个例子那样，对它进行缩进有助于提高可读性。 名称不是有效绑定名称或有效数字的属性必须加引号。

```
let descriptions = {
  work: "Went to work",
  "touched tree": "Touched a tree"
};
```

这意味着大括号在 JavaScript 中有两个含义。 在语句的开头，他们起始了一个语句块。 在任何其他位置，他们描述一个对象。 幸运的是，语句很少以花括号对象开始，因此这两者之间的不明确性并不是什么大问题。

读取一个不存在的属性就会产生`undefined`。

我们可以使用=运算符来给一个属性表达式赋值。如果该属性已经存在，那么这项操作就会替换原有的值。如果该属性不存在，则会在目标对象中新建一个属性。

简要回顾我们的绑定的触手模型 - 属性绑定也类似。 他们捕获值，但其他绑定和属性可能会持有这些相同的值。 你可以将对象想象成有任意数量触手的章鱼，每个触手上都有一个名字的纹身。

`delete`运算符切断章鱼的触手。 这是一个一元运算符，当应用于对象属性时，将从对象中删除指定的属性。 这不是一件常见的事情，但它是可能的。

```
let anObject = {left: 1, right: 2};
console.log(anObject.left);
// → 1
delete anObject.left;
console.log(anObject.left);
// → undefined
console.log("left" in anObject);
// → false
console.log("right" in anObject);
// → true
```

当应用于字符串和对象时，二元`in`运算符会告诉你该对象是否具有名称为它的属性。 将属性设置为`undefined`，和实际删除它的区别在于，在第一种情况下，对象仍然具有属性（它只是没有有意义的值），而在第二种情况下属性不再存在，`in`会返回`false`。

为了找出对象具有的属性，可以使用`Object.keys`函数。 你给它一个对象，它返回一个字符串数组 - 对象的属性名称。

```
console.log(Object.keys({x: 0, y: 0, z: 2}));
// → ["x", "y", "z"]
```

`Object.assign`函数可以将一个对象的所有属性复制到另一个对象中。

```
let objectA = {a: 1, b: 2};
bject.assign(objectA, {b: 3, c: 4});
console.log(objectA);
// → {a: 1, b: 3, c: 4}
```

然后，数组只是一种对象，专门用于存储对象序列。 如果你求解`typeof []`，它会产生`object`。 你可以看到它们是长而平坦的章鱼，它们的触手整齐排列，并以数字标记。

我们将雅克的日记表示为对象数组。

```
let journal = [
  {events: ["work", "touched tree", "pizza",
            "running", "television"],
   squirrel: false},
  {events: ["work", "ice cream", "cauliflower",
            "lasagna", "touched tree", "brushed teeth"],
   squirrel: false},
  {events: ["weekend", "cycling", "break", "peanuts",
            "beer"],
   squirrel: true},
  /* and so on... */
];
```

可变性

我们现在即将开始真正的编程。首先还有一个理论要理解。

我们看到对象值可以修改。千米你的章节讨论的值的类型（如数字，字符串和布尔值）都是不可变的 -- 这些类型的值不可能修改。你可以将它们组合起来并从它们派生新的值，但是当你采用特定的字符串值时，该值将始终保持不变。里面的文字不能改变。如果你有一个包含 `"cat"` 的字符串，其他代码不可能修改你的字符串中的一个字符，来使它变成 `"rat"` 。

对象的工作方式不同。你可以更改其属性，使单个对象值在不同时间具有不同的内容。

当我们有两个数字，120 和 120 时，我们可以将它们看作完全相同的数字，不管它们是否指向相同的物理位。使用对象时，拥有同一个对象的两个引用，和拥有包含相同属性的两个不同的对象，是有区别的。考虑下面的代码：

```
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};

console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false

object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

`object1` 和 `object2` 绑定持有相同对象，这就是为什么改变 `object1` 会改变 `object2` 的值。据说他们具有相同的身份。绑定 `object3` 指向一个不同的对象，它最初包含的属性与 `object1` 相同，但过着单独的生活。

绑定可以是可变的或不变的，但这与它们的值的行为方式是分开的。即使数值不变，你也可以使用 `let` 绑定来跟踪一个变化的数字，通过修改绑定所指向的值。与之类似，虽然对象的 `const` 绑定本身不可改变，并且始终指向相同对象，该对象的内容可能会改变。

```
const score = {visitors: 0, home: 0};
// This is okay
score.visitors = 1;
// This isn't allowed
score = {visitors: 1, home: 1};
```

当你用 JavaScript 的 `==` 运算符比较对象时，它按照身份进行比较：仅当两个对象的值严格相同时才产生 `true` 。 比较不同的对象会返回 `false` ，即使它们属性相同。 JavaScript 中没有内置的“深层”比较操作，它按照内容比较对象，但可以自己编写它（这是本章末尾的一个练习）。

松鼠人的记录

于是，雅克开始了他的 JavaScript 之旅，并搭建了用于保存每天记录的一套开发环境。

```
let journal = [];

function addEntry(events, squirrel) {
  journal.push({events, squirrel});
}
```

请注意添加到日记中的对象看起来有点奇怪。 它不像 `events:events` 那样声明属性，只是提供属性名称。 这是一个简写，意思一样 - 如果大括号中的属性名后面没有值，它的值来自相同名称的绑定。

那么，在每天晚上十点 -- 或者有时候是下一天的早晨，从它的书架顶部爬下来之后 -- 雅克记录了这一天。

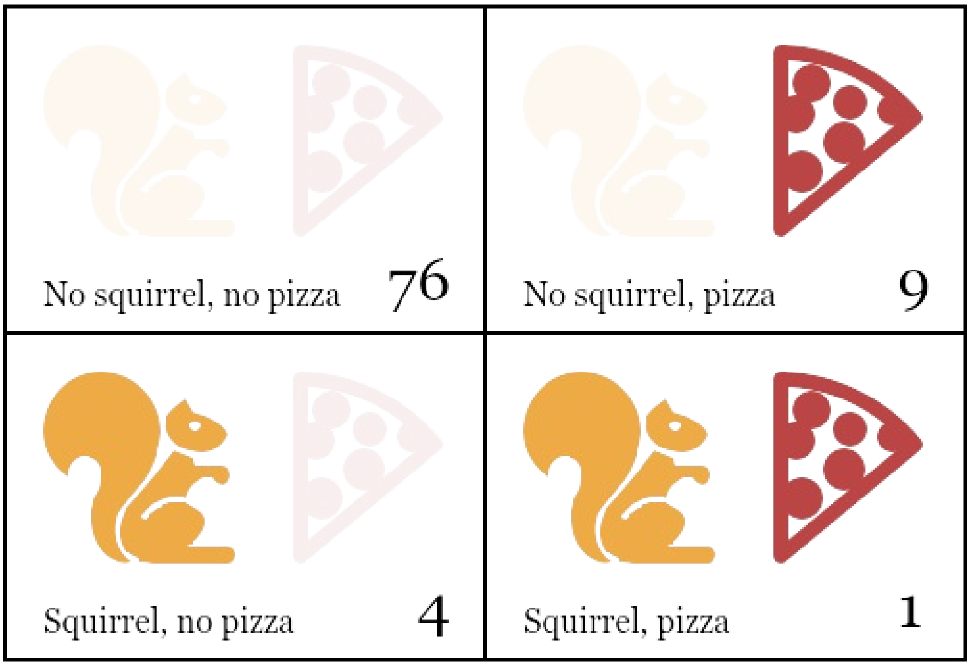
```
addEntry(["work", "touched tree", "pizza", "running",
          "television"], false);
addEntry(["work", "ice cream", "cauliflower", "lasagna",
          "touched tree", "brushed teeth"], false);
addEntry(["weekend", "cycling", "break", "peanuts",
          "beer"], true);
```

一旦他有了足够的数据点，他打算使用统计学来找出哪些事件可能与变成松鼠有关。

关联性是统计绑定之间的独立性的度量。 统计绑定与编程绑定不完全相同。 在统计学中，你通常会有一组度量，并且每个绑定都根据每个度量来测量。 绑定之间的相关性通常表示为从 -1 到 1 的值。 相关性为零意味着绑定不相关。 相关性为一表明两者完全相关 - 如果你知道一个，你也知道另一个。 负一意味着它们是完全相关的，但它们是相反的 - 当一个是真的时，另一个是假的。

为了计算两个布尔绑定之间的相关性度量，我们可以使用 phi 系数（ `ϕ` ）。 这是一个公式，输入为一个频率表格，包含观测绑定的不同组合的次数。 公式的输出是 -1 和 1 之间的数字。

我们可以将吃比萨的事件放在这样的频率表中，每个数字表示我们的度量中的组合的出现次数。



如果我们将那个表格称为 `n` ，我们可以用下列公式自己算 `ϕ` ：

$$\phi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet 1}n_{\bullet 0}}}$$

（如果你现在把这本书放下，专注于十年级数学课的可怕的再现，坚持住！我不打算用无休止的神秘符号折磨你 - 现在只有这一个公式。我们所做的就是把它变成 JavaScript。）

符号 `n01` 表明， 第一个绑定（松鼠）为假（0）时，第二个绑定（披萨）为真（1）。 在披萨表中， `n01` 是 9。

值 `n1` 表示所有度量之和，其中第一个绑定为 `true` ，在示例表中为 5。 同样， `n0` 表示所有度量之和，其中第二个绑定为假。

因此，我们以比萨表为例，除法线上方的部分（被除数）为 `1×76-9×4=40` ，而除法线下面的部分（除数）则是 `10×80×5×85` 的平方根，也就是 `√340000` 。计算结果为 `ϕ≈0.069` ，这个结果很小，因此吃比萨对是否变身成松鼠显然没有太大影响。

计算关联性

我们可以用包含 4 个元素的数组（ `[76，9，4，1]` ）来表示一张 2 乘 2 的表格。我们也可以使用其他表示方式，比如包含两个数组的数组，每个子数组又包含两个元素（ `[[76，9]，[4，1]]` ）。也可以使用一个对象，它包含一些属性，名为 `"11"` 和 `"01"` 。但

是，一维数组更为简单，也容易进行操作。我们可以将数组索引看成包含两个二进制位的数字，左边的（高位）数字表示绑定“是否变成松鼠”，右边的（低位）数字表示事件绑定。例如，若二进制数字为 10，表示雅克变成了松鼠，但事件并未发生（比如说吃比萨）。这种情况发生了 4 次。由于二进制数字 10 的十进制是 2，因此我们将其存储到数组中索引为 2 的位置上。

下面这个函数用于计算数组的系数 `ϕ`：

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}

console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

这将 `ϕ` 公式直接翻译成 JavaScript。 `Math.sqrt` 是平方根函数，由标准 JavaScript 环境中的 `Math` 对象提供。 我们必须在表格中添加两个字段来获取字段，例如 `n1` 因为行和或者列和不直接存储在我们的数据结构中。

雅克花了三个月的时间记录日志。在本章的代码沙箱（http://eloquentjavascript.net/code/）的下载文件中，用 `JOURNAL` 绑定存储了该结果数据集合。

若要从这篇记录中提取出某个特定事件的 2 乘 2 表格，我们首先需要循环遍历整个记录，并计算出与变身成松鼠相关事件发生的次数。

```
function hasEvent(event, entry) {
  return entry.events.indexOf(event) != -1;
}

function tableFor(event, journal) {
  let table = [0, 0, 0, 0];
  for (let i = 0; i < journal.length; i++) {
    let entry = journal[i], index = 0;
    if (entry.events.includes(event)) index += 1;
    if (entry.squirrel) index += 2;
    table[index] += 1;
  }
  return table;
}

console.log(tableFor("pizza", JOURNAL));
// → [76, 9, 4, 1]
```

数组拥有 `includes` 方法，检查给定值是否存在于数组中。 该函数使用它来确定，对于某一天，感兴趣的事件名称是否在事件列表中。

`tableFor` 中的循环体通过检查列表是否包含它感兴趣的特定事件，以及该事件是否与松鼠事件一起发生，来计算每个日记条目在表格中的哪个盒子。 然后循环对表中的正确盒子加一。

我们现在有了我们计算个体相关性的所需工具。 剩下的唯一一步，就是为记录的每种类型的事件找到关联，看看是否有什么明显之处。

数组循环

在 `tableFor` 函数中，有一个这样的循环：

```
for (let i = 0; i < JOURNAL.length; i++) {
  let entry = JOURNAL[i];
  // Do something with entry
}
```

这种循环在经典的 JavaScript 中很常见 - 遍历数组，一次一个元素会很常见，为此，你需要在数组长度上维护一个计数器，并依次选取每个元素。

在现代 JavaScript 中有一个更简单的方法来编写这样的循环。

```
for (let entry of JOURNAL) {
  console.log(`${entry.events.length} events.`);
}
```

当 `for` 循环看起来像这样，在绑定定义之后用 `of` 这个词时，它会遍历 `of` 之后的给定值的元素。 这不仅适用于数组，而且适用于字符串和其他数据结构。 我们将在第 6 章中讨论它的工作原理。

分析结果

我们需要计算数据集中发生的每种类型事件的相关性。 为此，我们首先需要寻找每种类型的事件。

```
function journalEvents(journal) {
  let events = [];
  for (let entry of journal) {
    for (let event of entry.events) {
      if (!events.includes(event)) {
        events.push(event);
      }
    }
  }
  return events;
}

console.log(journalEvents(JOURNAL));
// → ["carrot", "exercise", "weekend", "bread", …]
```

通过遍历所有事件，并将那些不在里面的事件添加到 `events` 数组中，该函数收集每种事件。

使用它，我们可以看到所有的相关性。

```
for (let event of journalEvents(JOURNAL)) {
  console.log(event + ":", phi(tableFor(event, JOURNAL)));
}
// → carrot:   0.0140970969
// → exercise: 0.0685994341
// → weekend:  0.1371988681
// → bread:   -0.0757554019
// → pudding: -0.0648203724
// and so on...
```

绝大多数相关系数都趋近于 0。显然，摄入胡萝卜、面包或布丁并不会导致变身成松鼠。但是似乎在周末变身成松鼠的概率更高。让我们过滤结果，来仅仅显示大于 0.1 或小于 -0.1 的相关性。

```
for (let event of journalEvents(JOURNAL)) {
  let correlation = phi(tableFor(event, JOURNAL));
  if (correlation > 0.1 || correlation < -0.1) {
    console.log(event + ":", correlation);
  }
}
// → weekend:        0.1371988681
// → brushed teeth: -0.3805211953
// → candy:          0.1296407447
// → work:          -0.1371988681
// → spaghetti:      0.2425356250
// → reading:        0.1106828054
// → peanuts:        0.5902679812
```

啊哈！这里有两个因素，其相关性明显强于其他因素。 吃花生对变成松鼠的几率有强烈的积极影响，而刷牙有显着的负面影响。

这太有意思了。让我们再仔细看看这些数据。

```
for (let entry of JOURNAL) {
  if (entry.events.includes("peanuts") &&
     !entry.events.includes("brushed teeth")) {
     entry.events.push("peanut teeth");
  }
}
console.log(phi(tableFor("peanut teeth", JOURNAL)));
// → 1
```

这是一个强有力的结果。 这种现象正好发生在雅克吃花生并且没有刷牙时。 如果他只是不注意口腔卫生，他从来没有注意到他的病痛。

知道这些之后，雅克完全停止吃花生，发现他的变身消失了。

几年来，雅克过得越来越好。 但是在某个时候他失去了工作。 因为他生活在一个糟糕的国家，没有工作就意味着没有医疗服务，所以他被迫在一个马戏团就业，在那里他扮演的是不可思议的松鼠人，在每场演出前都用花生酱塞满了它的嘴。

数组详解

在完成本章之前，我想向你介绍几个对象相关的概念。 我将首先介绍一些通常实用的数组方法。

我们在本章的前面已经了解了 `push` 和 `pop` 方法，分别用于在数组末尾添加或删除元素。相应地，在数组的开头添加或删除元素的方法分别是 `unshift` 和 `shift` 。

```
let todoList = [];
function remember(task) {
  todoList.push(task);
}
function getTask() {
  return todoList.shift();
}
function rememberUrgently(task) {
  todoList.unshift(task);
}
```

这个程序管理任务队列。 你通过调用 `remember("groceries")` ，将任务添加到队列的末尾，并且当你准备好执行某些操作时，可以调用 `getTask()` 从队列中获取（并删除）第一个项目。 `rememberUrgently` 函数也添加任务，但将其添加到队列的前面而不是队列的后面。

有一个与 `indexOf` 方法类似的方法，叫 `lastIndexOf` ，只不过 `indexOf` 从数组第一个元素向后搜索，而 `lastIndexOf` 从最后一个元素向前搜索。

```
console.log([1, 2, 3, 2, 1].indexOf(2));
// → 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2));
// → 3
```

`indexOf` 和 `lastIndexOf` 方法都有一个可选参数，可以用来指定搜索的起始位置。

另一个基本方法是 `slice` ，该方法接受一个起始索引和一个结束索引，然后返回数组中两个索引范围内的元素。起始索引元素包含在返回结果中，但结束索引元素不会包含在返回结果中。

```
console.log([0, 1, 2, 3, 4].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 4].slice(2));
// → [2, 3, 4]
```

如果没有指定结束索引， `slice` 会返回从起始位置之后的所有元素。你也可以省略起始索引来复制整个数组。

`concat` 方法可用于将数组粘在一起，来创建一个新数组，类似于 `+` 运算符对字符串所做的操作。

以下示例展示了 `concat` 和 `slice` 的作用。 它接受一个数组和一个索引，然后它返回一个新数组，该数组是原数组的副本，并且删除了给定索引处的元素：

```
function remove(array, index) {
  return array.slice(0, index)
    .concat(array.slice(index + 1));
}
console.log(remove(["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

如果你将`concat`传递给一个不是数组的参数，该值将被添加到新数组中，就像它是单个元素的数组一样。

字符串及其属性

我们可以调用字符串的`length`或`toUpperCase`这样的属性，但不能向字符串中添加任何新的属性。

```
let kim = "Kim";
kim.age = 88;
console.log(kim.age);
// → undefined
```

字符串、数字和布尔类型的值并不是对象，因此当你向这些值中添加属性时 JavaScript 并不会报错，但实际上你并没有将这些属性添加进去。前面说过，这些值是不变的，不能改变。

但这些类型包含一些内置属性。每个字符串中包含了若干方法供我们使用，最有用的方法可能就是`slice`和`indexOf`了，它们的功能与数组中的同名方法类似。

```
console.log("coconuts".slice(4, 7));
// → nut
console.log("coconut".indexOf("u"));
// → 5
```

一个区别是，字符串的`indexOf`可以搜索包含多个字符的字符串，而相应的数组方法仅查找单个元素。

```
console.log("one two three".indexOf("ee"));
// → 11
```

`trim`方法用于删除字符串中开头和结尾的空白符号（空格、换行符和制表符等符号）。

```
console.log("  okay \n ".trim());
// → okay
```

上一章中的`zeroPad`函数也作为方法存在。 它被称为`padStart`，接受所需的长度和填充字符作为参数。

```
console.log(String(6).padStart(3, "0"));
// → 006
```

你可以使用`split`，在另一个字符串的每个出现位置分割一个字符串，然后再用`join`把它连接在一起。

```
let sentence = "Secretarybirds specialize in stomping";
let words = sentence.split(" ");
console.log(words);
// → ["Secretarybirds", "specialize", "in", "stomping"]
console.log(words.join(". "));
// → Secretarybirds. specialize. in. stomping
```

可以用`repeat`方法重复一个字符串，该方法创建一个新字符串，包含原始字符串的多个副本，并将其粘在一起。

```
console.log("LA".repeat(3));
// → LALALA
```

我们已经看到了字符串类型的`length`属性。访问字符串中的单个字符，看起来像访问数组元素（有一个警告，我们将在第 5 章中讨论）。

```
let string = "abc";
console.log(string.length);
// → 3
console.log(string[1]);
// → b
```

剩余参数

一个函数可以接受任意数量的参数。例如，`Math.max`计算提供给它的参数的最大值。

为了编写这样一个函数，你需要在函数的最后一个参数之前放三个点，如下所示：

```
function max(...numbers) {
  let result = -Infinity;
  for (let number of numbers) {
    if (number > result) result = number;
  }
  return result;
}
console.log(max(4, 1, 9, -2));
// → 9
```

当这样的函数被调用时，剩余参数绑定一个数组，包含所有其它参数。如果之前有其他参数，它们的值不是该数组的一部分。当它是唯一的参数时，如`max`中那样，它将保存所有参数。

你可以使用类似的三点表示法，来使用参数数组调用函数。

```
let numbers = [5, 1, 7];
console.log(max(...numbers));
// → 7
```

这在函数调用中“展开”数组，并将其元素传递为单独的参数。 像`max(9, ...numbers, 2)'那样，可以包含像这样的数组以及其他参数。

方括号的数组表示法，同样允许三点运算符将另一个数组展开到新数组中：

```
let words = ["never", "fully"];
console.log(["will", ...words, "understand"]);
// → ["will", "never", "fully", "understand"]
```

Math对象

正如我们所看到的那样，`Math`对象中包含了许多与数字相关的工具函数，比如`Math.max`（求最大值）、`Math.min`（求最小值）和`Math.sqrt`（求平方根）。

`Math`对象被用作一个容器来分组一堆相关的功能。 只有一个`Math`对象，它作为一个值几乎没有用处。 相反，它提供了一个命名空间，使所有这些函数和值不必是全局绑定。

过多的全局绑定会“污染”命名空间。全局绑定越多，就越有可能一不小心把某些绑定的值覆盖掉。比如，我们可能想在程序中使用名为`max`的绑定，由于 JavaScript 将内置的`max`函数安全地放置在`Math`对象中，因此不必担心`max`的值会被覆盖。

当你去定义一个已经被使用的绑定名的时候，对于很多编程语言来说，都会阻止你这么做，至少会对这种行为发出警告。但是 JavaScript 不会，因此要小心这些陷阱。

让我们来继续了解`Math`对象。如果需要做三角运算，`Math`对象可以帮助到你，它包含`cos`（余弦）、`sin`（正弦）、`tan`（正切）和各自的反函数（`acos`、`asin`和`atan`）。`Math.PI`则表示数字`π`，或至少是 JavaScript 中的数字近似值。在传统的程序设计当中，常量均以大写来标注。

```
function randomPointOnCircle(radius) {
  let angle = Math.random() * 2 * Math.PI;
  return {x: radius * Math.cos(angle),
          y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

如果你对正弦或余弦不大熟悉，不必担心。我们会在第 13 章用到它们时，再做进一步解释。

在上面的示例代码中使用了`Math.random`。每次调用该函数时，会返回一个伪随机数，范围在 0（包括）到 1（不包括）之间。

```
console.log(Math.random());
// → 0.36993729369714856
console.log(Math.random());
// → 0.727367032552138
console.log(Math.random());
// → 0.40180766698904335
```

虽然计算机是确定性的机器，但如果给定相同的输入，它们总是以相同的方式作出反应 - 让它们产生随机显示的数字是可能的。 为此，机器会维护一些隐藏的值，并且每当你请求一个新的随机数时，它都会对该隐藏值执行复杂的计算来创建一个新值。 它存储一个新值并返回从中派生的一些数字。 这样，它可以以随机的方式产生新的，难以预测的数字。

如果我们想获取一个随机的整数而非小数，可以使用 `Math.floor` （向下取整到与当前数字最接近的整数）来处理 `Math.random` 的结果。

```
console.log(Math.floor(Math.random() * 10));
// → 2
```

将随机数乘以 10 可以得到一个在 0 到 10 之间的数字。由于 `Math.floor` 是向下取整，因此该函数会等概率地取到 0 到 9 中的任何一个数字。

还有两个函数，分别是 `Math.ceil` （向上取整）和 `Math.round` （四舍五入）。以及 `Math.abs` ，它取数字的绝对值，这意味着它反转了负值，但保留了正值。

解构

让我们暂时回顾 `phi` 函数：

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}
```

这个函数难以阅读的原因之一，是我们有一个指向数组的绑定，但我们更愿意拥有数组的元素的绑定，即 `let n00 = table [0]` 以及其他。 幸运的是，有一种简洁的方法可以在 JavaScript 中执行此操作。

```
function phi([n00, n01, n10, n11]) {
  return (n11 * n00 - n10 * n01) /
    Math.sqrt((n10 + n11) * (n00 + n01) *
              (n01 + n11) * (n00 + n10));
}
```

这也适用于由 `let` ， `var` 或 `const` 创建的绑定。 如果你知道要绑定的值是一个数组，则可以使用方括号来“向内查看”该值，并绑定其内容。

类似的技巧适用于对象，使用大括号代替方括号。

```
let {name} = {name: "Faraji", age: 23};
console.log(name);
// → Faraji
```

请注意，如果尝试解构 `null` 或 `undefined` ，则会出现错误，就像直接尝试访问这些值的属性一样。

JSON

因为属性只是捕获了它们的值，而不是包含它们，对象和数组在计算机的内存中储存为字节序列，存放它们的内容的地址（内存中的位置）。因此，包含另一个数组的数组，（至少）由两个内存区域组成，一个用于内部数组，另一个用于外部数组，（除了其它东西之外）其中包含表示内部数组位置的二进制数。

如果你想稍后将数据保存到文件中，或者通过网络将其发送到另一台计算机，则必须以某种方式，将这些内存地址的线团转换为可以存储或发送的描述。我想你应该把你的整个计算机内存，连同你感兴趣的值的地址一起发送，但这似乎并不是最好的方法。

我们可以做的是序列化数据。这意味着它被转换为扁平的描述。流行的序列化格式称为 JSON（发音为“Jason”），它代表 JavaScript Object Notation（JavaScript 对象表示法）。它被广泛用作 Web 上的数据存储和通信格式，即使在 JavaScript 以外的语言中也是如此。

JSON 看起来像 JavaScript 的数组和对象的表示方式，但有一些限制。所有属性名都必须用双引号括起来，并且只允许使用简单的数据表达式 - 没有函数调用，绑定或任何涉及实际计算的内容。JSON 中不允许注释。

表示为 JSON 数据时，日记条目可能看起来像这样

```
{
  "squirrel": false,
  "events": ["work", "touched tree", "pizza", "running"]
}
```

JavaScript 为我们提供了函数 `JSON.stringify` 和 `JSON.parse` ，来将数据转换为这种格式，以及从这种格式转换。第一个函数接受 JavaScript 值并返回 JSON 编码的字符串。第二个函数接受这样的字符串并将其转换为它编码的值。

```
let string = JSON.stringify({squirrel: false,
                             events: ["weekend"]});
console.log(string);
// → {"squirrel":false,"events":["weekend"]}
console.log(JSON.parse(string).events);
// → ["weekend"]
```

本章小结

对象和数组（一种特殊对象）可以将几个值组合起来形成一个新的值。理论上说，我们可以将一组相关的元素打包成一个对象，并通过这个对象来访问这些元素，以避免管理那些支离破碎的元素。

在 JavaScript 中，除了 `null` 和 `undefined` 以外，绝大多数的值都含有属性。我们可以用 `value.prop` 或 `value["prop"]` 来访问属性。对象使用名称来定义和存储一定数量的属性。另外，数组中通常会包含不同数量的值，并使用数字（从 0 开始）作为这些值的属性。

在数组中有一些具名属性，比如 `length` 和一些方法。方法是作为属性存在的函数，常常作用于其所属的值。

你可以使用特殊类型的 `for` 循环 `for (let element of array)` 来迭代数组。

习题

范围的和

在本书的前言中，提到过一种很好的计算固定范围内数字之和的方法：

```
console.log(sum(range(1, 10)));
```

编写一个 `range` 函数，接受两个参数：`start` 和 `end`，然后返回包含 `start` 到 `end`（包括 `end`）之间的所有数字。

接着，编写一个 `sum` 函数，接受一个数字数组，并返回所有数字之和。运行示例程序，检查一下结果是不是 55。

附加题是修改 `range` 函数，接受第 3 个可选参数，指定构建数组时的步长（`step`）。如果没有指定步长，构建数组时，每步增长 1，和旧函数行为一致。调用函数 `range(1, 10, 2)`，应该返回 `[1, 3, 5, 7, 9]`。另外确保步数值为负数时也可以正常工作，因此 `range(5, 2, -1)` 应该产生 `[5, 4, 3, 2]`。

```
// Your code here.

console.log(range(1, 10));
// → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
console.log(range(5, 2, -1));
// → [5, 4, 3, 2]
console.log(sum(range(1, 10)));
// → 55
```

逆转数组

数组有一个 `reverse` 方法，它可以逆转数组中元素的次序。在本题中，编写两个函数，`reverseArray` 和 `reverseArrayInPlace`。第一个函数 `reverseArray` 接受一个数组作为参数，返回一个新数组，并逆转新数组中的元素次序。第二个函数 `reverseArrayInPlace` 与第一个函数的功能相同，但是直接将数组作为参数进行修改来，逆转数组中的元素次序。两者都不能使用标准的 `reverse` 方法。

回想一下，在上一章中关于副作用和纯函数的讨论，哪个函数的写法的应用场景更广？哪个执行得更快？

```
// Your code here.

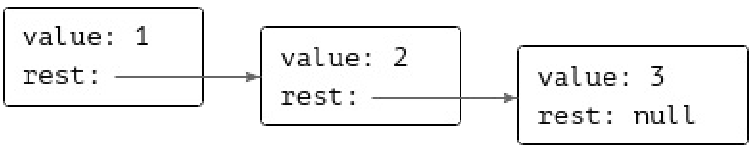
console.log(reverseArray(["A", "B", "C"]));
// → ["C", "B", "A"];
let arrayValue = [1, 2, 3, 4, 5];
reverseArrayInPlace(arrayValue);
console.log(arrayValue);
// → [5, 4, 3, 2, 1]
```

实现列表

对象作为一个值的容器，它可以用来构建各种各样的数据结构。有一种通用的数据结构叫作列表（list）（不要与数组混淆）。列表是一种嵌套对象集合，第一个对象拥有第二个对象的引用，而第二个对象有第三个对象的引用，依此类推。

```
let list = {
  value: 1,
  rest: {
    value: 2,
    rest: {
      value: 3,
      rest: null
    }
  }
};
```

最后产生的对象形成了一条链，如下图所示：



使用列表的一个好处是，它们之间可以共享相同的子列表。举个例子，如果我们新建了两个值：`{value: 0，result: list}` 和 `{value: -1，result: list}`（`list` 引用了我们前面定义的绑定）。这是两个独立的列表，但它们之间却共享了同一个数据结构，该数据结构包含列表末尾的三个元素。而且我们前面定义的 `list` 仍然是包含三个元素的列表。

编写一个函数 `arrayToList`，当给定参数 `[1, 2, 3]` 时，建立一个和示例相似的数据结构。然后编写一个 `listToArray` 函数，将列表转换成数组。再编写一个工具函数 `prepend`，接受一个元素和一个列表，然后创建一个新的列表，将元素添加到输入列表的开头。最后编写一个函

数 `nth`，接受一个列表和一个数，并返回列表中指定位置的元素，如果该元素不存在则返回 `undefined`。

如果你觉得这都不是什么难题，那么编写一个递归版本的 `nth` 函数。

```
// Your code here.

console.log(arrayToList([10, 20]));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(listToArray(arrayToList([10, 20, 30])));
// → [10, 20, 30]
console.log(prepend(10, prepend(20, null)));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(nth(arrayToList([10, 20, 30]), 1));
// → 20
```

深层比较

`==` 运算符可以判断对象是否相等。但有些时候，你希望比较的是对象中实际属性的值。

编写一个函数 `deepEqual`，接受两个参数，若两个对象是同一个值或两个对象中有相同属性，且使用 `deepEqual` 比较属性值均返回 `true` 时，返回 `true`。

为了弄清楚通过身份（使用 `===` 运算符）还是其属性比较两个值，可以使用 `typeof` 运算符。如果对两个值使用 `typeof` 均返回 `"object"`，则说明你应该进行深层比较。但需要考虑一个例外的情况：由于历史原因，`typeof null` 也会返回 `"object"`。

当你需要查看对象的属性来进行比较时，`Object.keys` 函数将非常有用。

```
// Your code here.

let obj = {here: {is: "an"}, object: 2};
console.log(deepEqual(obj, obj));
// → true
console.log(deepEqual(obj, {here: 1, object: 2}));
// → false
console.log(deepEqual(obj, {here: {is: "an"}, object: 2}));
// → true
```

五、高阶函数

> 原文：Higher-Order Functions
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> Tzu-li and Tzu-ssu were boasting about the size of their latest programs. ‘Two-hundred thousand lines,’ said Tzu-li, ‘not counting comments!’ Tzu-ssu responded, ‘Pssh, mine is almost a million lines already.’ Master Yuan-Ma said, ‘My best program has five hundred lines.’ Hearing this, Tzu-li and Tzu-ssu were enlightened.
>
> Master Yuan-Ma，《The Book of Programming》
>
> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.
>
> C.A.R. Hoare，1980 ACM Turing Award Lecture



开发大型程序通常需要耗费大量财力和物力，这绝不仅仅是因为构建程序所花费时间的问题。大型程序的复杂程度总是很高，而这些复杂性也会给开发人员带来不少困扰，而程序错误或 bug 往往就是这些时候引入的。大型程序为这些 bug 提供了良好的藏身之所，因此我们更加难以在大型程序中找到它们。

让我们简单回顾一下前言当中的两个示例。其中第一个程序包含了 6 行代码并可以直接运行。

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

第二个程序则依赖于外部函数才能执行，且只有一行代码。

```
console.log(sum(range(1, 10)));
```

哪一个程序更有可能含有 bug 呢？

如果算上 `sum` 和 `range` 两个函数的代码量，显然第二个程序的代码量更大。不过，我仍然觉得第二个程序包含 bug 的可能性比第一个程序低。

之所以这么说的原因是，第二个程序编写的代码很好地表达了我们期望解决的问题。对于计算一组数字之和这个操作来说，我们关注的是计算范围和求和运算，而不是循环和计数。

`sum` 和 `range` 这两个函数定义的操作当然会包含循环、计数和其他一些操作。但相比于将这些代码直接写到一起，这种表述方式更为简单，同时也易于避免错误。

抽象

在程序设计中，我们把这种编写代码的方式称为抽象。抽象可以隐藏底层的实现细节，从更高（或更加抽象）的层次看待我们要解决的问题。

举个例子，比较一下这两份豌豆汤的食谱：

按照每人一杯的量将脱水豌豆放入容器中。倒水直至浸没豌豆，然后至少将豌豆浸泡 12 个小时。将豌豆从水中取出沥干，倒入煮锅中，按照每人四杯水的量倒入水。将食材盖满整个锅底，并慢煮 2 个小时。按照每人半个的量加入洋葱，用刀切片，然后放入豌豆中。按照每人一根的量加入芹菜，用刀切片，然后放入豌豆当中。按照每人一根的量放入胡萝卜，用刀切片，然后放入豌豆中。最后一起煮 10 分钟以上即可。

第二份食谱：

一个人的量：一杯脱水豌豆、半个切好的洋葱、一根芹菜和一根胡萝卜。

将豌豆浸泡 12 个小时。按照每人四杯水的量倒入水，然后用文火煨 2 个小时。加入切片的蔬菜，煮 10 分钟以上即可。

相比第一份食谱，第二份食谱更简短且更易于理解。但你需要了解一些有关烹调的术语：浸泡、煨、切片，还有蔬菜。

在编程的时候，我们不能期望所有功能都是现成的。因此，你可能就会像第一份食谱那样编写你的程序，逐个编写计算机需要执行的代码和步骤，而忽略了这些步骤之上的抽象概念。

在编程时，注意你的抽象级别什么时候过低，是一项非常有用的技能。

重复的抽象

我们已经了解的普通函数就是一种很好的构建抽象的工具。但有些时候，光有函数也不一定能够解决我们的问题。

程序以给定次数执行某些操作很常见。 你可以为此写一个`for`循环，就像这样：

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

我们是否能够将“做某件事`N`次”抽象为函数？ 编写一个调用`console.log` `N`次的函数是很容易的。

```
function repeatLog(n) {
  for (let i = 0; i < n; i++) {
    console.log(i);
  }
}
```

但如果我们想执行打印数字以外的操作该怎么办呢？我们可以使用函数来定义我们想做的事，而函数也是值，因此我们可以将期望执行的操作封装成函数，然后传递进来。

```
function repeat(n, action) {
  for (let i = 0; i < n; i++) {
    action(i);
  }
}

repeat(3, console.log);
// → 0
// → 1
// → 2
```

你不必将预定义的函数传递给`repeat`。 通常情况下，你希望原地创建一个函数值。

```
let labels = [];
repeat(5, i => {
  labels.push(`Unit ${i + 1}`);
});
console.log(labels);
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

这个结构有点像 `for` 循环 - 它首先描述了这种循环，然后提供了一个主体。 但是，主体现在写为一个函数值，它被包裹在 `repeat` 调用的括号中。 这就是它必须用右小括号和右大括号闭合的原因。 在这个例子中，主体是单个小表达式，你也可以省略大括号并将循环写成单行。

高阶函数

如果一个函数操作其他函数，即将其他函数作为参数或将函数作为返回值，那么我们可以将其称为高阶函数。因为我们已经看到函数就是一个普通的值，那么高阶函数也就不是什么稀奇的概念了。高阶这个术语来源于数学，在数学当中，函数和值的概念有着严格的区分。

我们可以使用高阶函数对一系列操作和值进行抽象。高阶函数有多种表现形式。比如你可以使用高阶函数来新建另一些函数。

```
function greaterThan(n) {
  return m => m > n;
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

你也可以使用高阶函数来修改其他的函数。

```
function noisy(f) {
  return (...args) => {
    console.log("calling with", args);
    let result = f(...args);
    console.log("called with", args, ", returned", result);
    return result;
  };
}
noisy(Math.min)(3, 2, 1);
// → calling with [3, 2, 1]
// → called with [3, 2, 1] , returned 1
```

你甚至可以使用高阶函数来实现新的控制流。

```
function unless(test, then) {
  if (!test) then();
}
repeat(3, n => {
  unless(n % 2 == 1, () => {
    console.log(n, "is even");
  });
});
// → 0 is even
// → 2 is even
```

有一个内置的数组方法，`forEach`，它提供了类似 `for/of` 循环的东西，作为一个高阶函数。

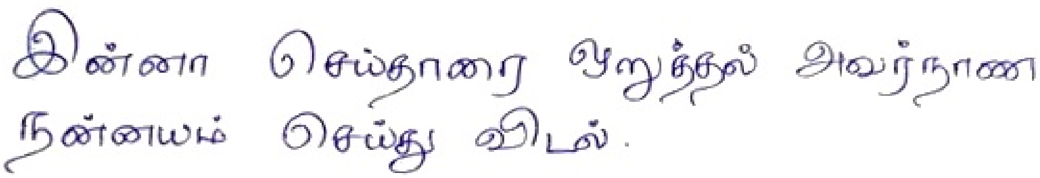
```
["A", "B"].forEach(l => console.log(l));
// → A
// → B
```

脚本数据集

数据处理是高阶函数表现突出的一个领域。 为了处理数据，我们需要一些真实数据。 本章将使用脚本书写系统的数据集，例如拉丁文，西里尔文或阿拉伯文。

请记住第 1 章中的 Unicode，该系统为书面语言中的每个字符分配一个数字。 大多数这些字符都与特定的脚本相关联。 该标准包含 140 个不同的脚本 - 81 个今天仍在使用，59 个是历史性的。

虽然我只能流利地阅读拉丁字符，但我很欣赏这样一个事实，即人们使用其他至少 80 种书写系统来编写文本，其中许多我甚至不认识。 例如，以下是泰米尔语手写体的示例。



示例数据集包含 Unicode 中定义的 140 个脚本的一些信息。 本章的编码沙箱中提供了 `SCRIPTS` 绑定。 该绑定包含一组对象，其中每个对象都描述了一个脚本。

```
{
  name: "Coptic",
  ranges: [[994, 1008], [11392, 11508], [11513, 11520]],
  direction: "ltr",
  year: -200,
  living: false,
  link: "https://en.wikipedia.org/wiki/Coptic_alphabet"
}
```

这样的对象会告诉你脚本的名称，分配给它的 Unicode 范围，书写方向，（近似）起始时间，是否仍在使用以及更多信息的链接。 方向可以是从左到右的 `"ltr"`，从右到左的 `"rtl"`（阿拉伯语和希伯来语文字的写法），或者从上到下的 `"ttb"`（蒙古文的写法）。

`ranges` 属性包含 Unicode 字符范围数组，每个数组都有两元素，包含下限和上限。 这些范围内的任何字符码都会分配给脚本。 下限是包括的（代码 994 是一个科普特字符），并且上限排除在外（代码 1008 不是）。

数组过滤

为了找到数据集中仍在使用的脚本，以下函数可能会有所帮助。 它过滤掉数组中未通过测试的元素：

```
function filter(array, test) {
  let passed = [];
  for (let element of array) {
    if (test(element)) {
      passed.push(element);
    }
  }
  return passed;
}

console.log(filter(SCRIPTS, script => script.living));
// → [{name: "Adlam", …}, …]
```

该函数使用名为 `test` 的参数（一个函数值）填充计算中的“间隙” - 决定要收集哪些元素的过程。

需要注意的是，`filter` 函数并没有从当前数组中删除元素，而是新建了一个数组，并将满足条件的元素存入新建的数组中。这个函数是一个“纯函数”，因为该函数并未修改给定的数组。

与 `forEach` 一样，`filter` 函数也是标准的数组方法。本例中定义的函数只是用于展示内部实现原理。今后我们会使用以下方法来过滤数据：

```
console.log(SCRIPTS.filter(s => s.direction == "ttb"));
// → [{name: "Mongolian", …}, …]
```

使用 `map` 函数转换数组

假设我们已经通过某种方式过滤了 `SCRIPTS` 数组，生成一个用于表示脚本的信息数组。但我们想创建一个包含名称的数组，因为这样更加易于检查。

`map` 方法对数组中的每个元素调用函数，然后利用返回值来构建一个新的数组，实现转换数组的操作。新建数组的长度与输入的数组一致，但其中的内容却通过对每个元素调用的函数“映射”成新的形式。

```
function map(array, transform) {
  let mapped = [];
  for (let element of array) {
    mapped.push(transform(element));
  }
  return mapped;
}

let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");
console.log(map(rtlScripts, s => s.name));
// → ["Adlam", "Arabic", "Imperial Aramaic", …]
```

与 `forEach` 和 `filter` 一样，`map` 也是标准的数组方法。

使用 `reduce` 汇总数据

与数组有关的另一个常见事情是从它们中计算单个值。 我们的递归示例，汇总了一系列数字，就是这样一个例子。 另一个例子是找到字符最多的脚本。

表示这种模式的高阶操作称为归约（reduce）（有时也称为折叠（fold））。 它通过反复从数组中获取单个元素，并将其与当前值合并来构建一个值。 在对数字进行求和时，首先从数字零开始，对于每个元素，将其与总和相加。

`reduce`函数包含三个参数：数组、执行合并操作的函数和初始值。该函数没有`filter`和`map`那样直观，所以仔细看看：

```
function reduce(array, combine, start) {
  let current = start;
  for (let element of array) {
    current = combine(current, element);
  }
  return current;
}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));
// → 10
```

数组中有一个标准的`reduce`方法，当然和我们上面看到的那个函数一致，可以简化合并操作。如果你的数组中包含多个元素，在调用`reduce`方法的时候忽略了`start`参数，那么该方法将会使用数组中的第一个元素作为初始值，并从第二个元素开始执行合并操作。

```
console.log([1, 2, 3, 4].reduce((a, b) => a + b));
// → 10
```

为了使用`reduce`（两次）来查找字符最多的脚本，我们可以这样写：

```
function characterCount(script) {
  return script.ranges.reduce((count, [from, to]) => {
    return count + (to - from);
  }, 0);
}

console.log(SCRIPTS.reduce((a, b) => {
  return characterCount(a) < characterCount(b) ? b : a;
}));
// → {name: "Han", …}
```

`characterCount`函数通过累加范围的大小，来减少分配给脚本的范围。 请注意归约器函数的参数列表中使用的解构。 `reduce'的第二次调用通过重复比较两个脚本并返回更大的脚本，使用它来查找最大的脚本。

Unicode 标准分配了超过 89,000 个字符给汉字脚本，它成为数据集中迄今为止最大的书写系统。 汉字是一种（有时）用于中文，日文和韩文的文字。 这些语言共享很多字符，尽管他们倾向于以不同的方式写它们。 （基于美国的）Unicode 联盟决定将它们看做一个单独的书写系统来保存字符码。 这被称为中日韩越统一表意文字（Han unification），并且仍然使一些人非常生气。

可组合性

考虑一下，我们怎样才可以在不使用高阶函数的情况下，编写以上示例（找到最大的脚本）？代码没有那么糟糕。

```
let biggest = null;
for (let script of SCRIPTS) {
  if (biggest == null ||
      characterCount(biggest) < characterCount(script)) {
    biggest = script;
  }
}
console.log(biggest);
// → {name: "Han", …}
```

这段代码中多了一些绑定，虽然多了两行代码，但代码逻辑还是很容易让人理解的。

当你需要组合操作时，高阶函数的价值就突显出来了。举个例子，我们编写一段代码，找出数据集中男人和女人的平均年龄。

```
function average(array) {
  return array.reduce((a, b) => a + b) / array.length;
}

console.log(Math.round(average(
  SCRIPTS.filter(s => s.living).map(s => s.year))));
// → 1185
console.log(Math.round(average(
  SCRIPTS.filter(s => !s.living).map(s => s.year))));
// → 209
```

因此，Unicode 中的死亡脚本，平均比活动脚本更老。 这不是一个非常有意义或令人惊讶的统计数据。 但是我希望你会同意，用于计算它的代码不难阅读。 你可以把它看作是一个流水线：我们从所有脚本开始，过滤出活动的（或死亡的）脚本，从这些脚本中抽出时间，对它们进行平均，然后对结果进行四舍五入。

你当然也可以把这个计算写成一个大循环。

```
let total = 0, count = 0;
for (let script of SCRIPTS) {
  if (script.living) {
    total += script.year;
    count += 1;
  }
}
console.log(Math.round(total / count));
// → 1185
```

但很难看到正在计算什么以及如何计算。 而且由于中间结果并不表示为一致的值，因此将“平均值”之类的东西提取到单独的函数中，需要更多的工作。

就计算机实际在做什么而言，这两种方法也是完全不同的。第一个在运行`filter`和`map`的时候会建立新的数组，而第二个只会计算一些数字，从而减少工作量。你通常可以采用可读的方法，但是如果你正在处理巨大的数组，并且多次执行这些操作，那么抽象风格的加速就是值得的。

字符串和字符码

这个数据集的一种用途是确定一段文本所使用的脚本。我们来看看执行它的程序。

请记住，每个脚本都有一组与其相关的字符码范围。所以给定一个字符码，我们可以使用这样的函数来找到相应的脚本（如果有的话）：

```
function characterScript(code) {
  for (let script of SCRIPTS) {
    if (script.ranges.some(([from, to]) => {
      return code >= from && code < to;
    })) {
      return script;
    }
  }
  return null;
}

console.log(characterScript(121));
// → {name: "Latin", …}
```

`some`方法是另一个高阶函数。它需要一个测试函数，并告诉你该函数是否对数组中的任何元素返回`true`。

但是，我们如何获得字符串中的字符码？

在第一章中，我提到 JavaScript 字符串被编码为一个 16 位数字的序列。这些被称为代码单元。一个 Unicode 字符代码最初应该能放进这样一个单元（它给你超 65,000 个字符）。后来人们发现它不够用了，很多人避开了为每个字符使用更多内存的需求。为了解决这些问题，人们发明了 UTF-16，JavaScript 字符串使用的格式。它使用单个 16 位代码单元描述了大多数常见字符，但是为其他字符使用一对两个这样的单元。

今天 UTF-16 通常被认为是一个糟糕的主意。它似乎总是故意设计来引起错误。很容易编写程序，假装代码单元和字符是一个东西。如果你的语言不使用两个单位的字符，显然能正常工作。但只要有人试图用一些不太常见的中文字符来使用这样的程序，就会中断。幸运的是，随着 emoji 符号的出现，每个人都开始使用两个单元的字符，处理这些问题的负担更加分散。

```
// Two emoji characters, horse and shoe
let horseShoe = "\ud83d\udc34\ud83d\udc5f";
console.log(horseShoe.length);
// → 4
console.log(horseShoe[0]);
// → (Invalid half-character)
console.log(horseShoe.charCodeAt(0));
// → 55357 (Code of the half-character)
console.log(horseShoe.codePointAt(0));
// → 128052 (Actual code for horse emoji)
```

JavaScript的`charCodeAt`方法为你提供了一个代码单元，而不是一个完整的字符代码。稍后添加的`codePointAt`方法确实提供了完整的 Unicode 字符。所以我们可以使用它从字符串中获取字符。但传递给`codePointAt`的参数仍然是代码单元序列的索引。因此，要运行字符串中的所有字符，我们仍然需要处理一个字符占用一个还是两个代码单元的问题。

在上一章中，我提到`for/of`循环也可以用在字符串上。像`codePointAt`一样，这种类型的循环，是在人们敏锐地意识到 UTF-16 的问题的时候引入的。当你用它来遍历一个字符串时，它会给你真正的字符，而不是代码单元。

```
let roseDragon = "\ud83c\udf45\ud83d\udc09";
for (let char of roseDragon) {
  console.log(char);
// → (emoji rose)
// → (emoji dragon)
```

如果你有一个字符（它是一个或两个代码单元的字符串），你可以使用`codePointAt(0)`来获得它的代码。

识别文本

我们有了`characterScript`函数和一种正确遍历字符的方法。下一步将是计算属于每个脚本的字符。下面的计数抽象会很实用：

```
function countBy(items, groupName) {
  let counts = [];
  for (let item of items) {
    let name = groupName(item);
    let known = counts.findIndex(c => c.name == name);
    if (known == -1) {
      counts.push({name, count: 1});
    } else {
      counts[known].count++;
    }
  }
  return counts;
}

console.log(countBy([1, 2, 3, 4, 5], n => n > 2));
// → [{name: false, count: 2}, {name: true, count: 3}]
```

`countBy` 函数需要一个集合（我们可以用 `for/of` 来遍历的任何东西）以及一个函数，它计算给定元素的组名。 它返回一个对象数组，每个对象命名一个组，并告诉你该组中找到的元素数量。

它使用另一个数组方法 `findIndex` 。 这个方法有点像 `indexOf` ，但它不是查找特定的值，而是查找给定函数返回 `true` 的第一个值。 像 `indexOf` 一样，当没有找到这样的元素时，它返回 -1。

使用 `countBy` ，我们可以编写一个函数，告诉我们在一段文本中使用了哪些脚本。

```
function textScripts(text) {
  let scripts = countBy(text, char => {
    let script = characterScript(char.codePointAt(0));
    return script ? script.name : "none";
  }).filter(({name}) => name != "none");

  let total = scripts.reduce((n, {count}) => n + count, 0);
  if (total == 0) return "No scripts found";

  return scripts.map(({name, count}) => {
    return `${Math.round(count * 100 / total)}% ${name}`;
  }).join(", ");
}

console.log(textScripts('英国的狗说"woof", 俄罗斯的狗说"тяв"'));
// → 61% Han, 22% Latin, 17% Cyrillic
```

该函数首先按名称对字符进行计数，使用 `characterScript` 为它们分配一个名称，并且对于不属于任何脚本的字符，回退到字符串 `"none"` 。 `filter` 调用从结果数组中删除 `"none"` 的条目，因为我们对这些字符不感兴趣。

为了能够计算百分比，我们首先需要属于脚本的字符总数，我们可以用 `reduce` 来计算。 如果没有找到这样的字符，该函数将返回一个特定的字符串。 否则，它使用 `map` 将计数条目转换为可读的字符串，然后使用 `join` 合并它们。

本章小结

能够将函数值传递给其他函数，是 JavaScript 的一个非常有用的方面。 它允许我们编写函数，用它们中的“间隙”对计算建模。 调用这些函数的代码，可以通过提供函数值来填补间隙。

数组提供了许多有用的高阶方法。 你可以使用 `forEach` 来遍历数组中的元素。 `filter` 方法返回一个新数组，只包含通过谓词函数的元素。 通过将函数应用于每个元素的数组转换，使用 `map` 来完成。 你可以使用 `reduce` 将数组中的所有元素合并为一个值。 `some` 方法测试任何元素是否匹配给定的谓词函数。 `findIndex` 找到匹配谓词的第一个元素的位置。

习题

展开

联合使用`reduce`方法和`concat`方法，将一个数组的数组“展开”成一个单个数组，包含原始数组的所有元素。

```
let arrays = [[1, 2, 3], [4, 5], [6]];
// Your code here.
// → [1, 2, 3, 4, 5, 6]
```

你自己的循环

编写一个高阶函数`loop`，提供类似`for`循环语句的东西。它接受一个值，一个测试函数，一个更新函数和一个主体函数。每次迭代中，它首先在当前循环值上运行测试函数，并在返回`false`时停止。然后它调用主体函数，向其提供当前值。最后，它调用`update`函数来创建一个新的值，并从头开始。

定义函数时，可以使用常规循环来执行实际循环。

```
// Your code here.

loop(3, n => n > 0, n => n - 1, console.log);
// → 3
// → 2
// → 1
```

`every`

类似于`some`方法，数组也有`every`方法。当给定函数对数组中的每个元素返回`true`时，此函数返回`true`。在某种程度上，`some`是作用于数组的`||`运算符的一个版本，`every`就像`&&`运算符。

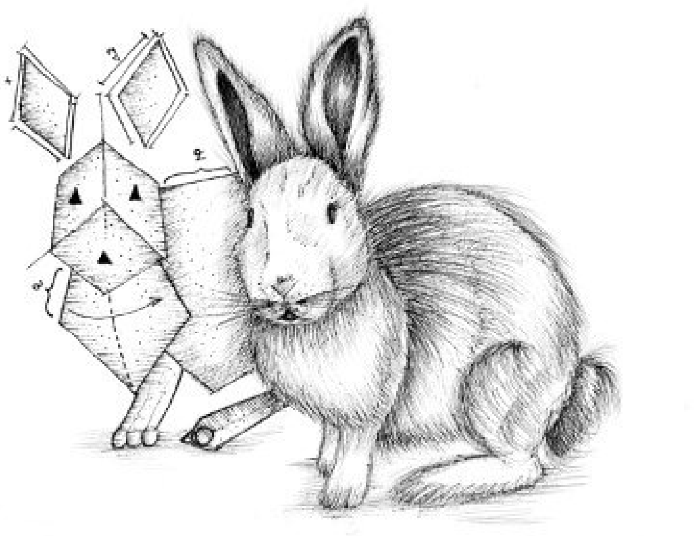
将`every`实现为一个函数，接受一个数组和一个谓词函数作为参数。编写两个版本，一个使用循环，另一个使用`some`方法。

```
function every(array, test) {
  // Your code here.
}

console.log(every([1, 3, 5], n => n < 10));
// → true
console.log(every([2, 4, 16], n => n < 10));
// → false
console.log(every([], n => n < 10));
// → true
```

六、对象的秘密

> 原文：The Secret Life of Objects
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 抽象数据类型是通过编写一种特殊的程序来实现的，该程序根据可在其上执行的操作来定义类型。
>
> Barbara Liskov，《Programming with Abstract Data Types》



第 4 章介绍了 JavaScript 的对象（object）。 在编程文化中，我们有一个名为面向对象编程（OOP）的东西，这是一组技术，使用对象（和相关概念）作为程序组织的中心原则。

虽然没有人真正同意其精确定义，但面向对象编程已经成为了许多编程语言的设计，包括 JavaScript 在内。 本章将描述这些想法在 JavaScript 中的应用方式。

封装

面向对象编程的核心思想是将程序分成小型片段，并让每个片段负责管理自己的状态。

通过这种方式，一些程序片段的工作方式的知识可以局部保留。 从事其他方面的工作的人，不必记住甚至不知道这些知识。 无论什么时候这些局部细节发生变化，只需要直接更新其周围的代码。

这种程序的不同片段通过接口（interface），函数或绑定的有限集合交互，它以更抽象的级别提供有用的功能，并隐藏它的精确实现。

这些程序片段使用对象建模。 它们的接口由一组特定的方法（method）和属性（property）组成。 接口的一部分的属性称为公共的（public）。 其他外部代码不应该接触属性的称为私有的（private）。

许多语言提供了区分公共和私有属性的方法，并且完全防止外部代码访问私有属性。 JavaScript 再次采用极简主义的方式，没有。 至少目前还没有 - 有个正在开展的工作，将其添加到该语言中。

即使这种语言没有内置这种区别，JavaScript 程序员也成功地使用了这种想法。 通常，可用的接口在文档或数字一中描述。 在属性名称的的开头经常会放置一个下划线（`_`）字符，来表明这些属性是私有的。

将接口与实现分离是一个好主意。 它通常被称为封装（encapsulation）。

方法

方法不过是持有函数值的属性。 这是一个简单的方法：

```
let rabbit = {};
rabbit.speak = function(line) {
  console.log(`The rabbit says '${line}'`);
};

rabbit.speak("I'm alive.");
// → The rabbit says 'I'm alive.'
```

方法通常会在对象被调用时执行一些操作。将函数作为对象的方法调用时，会找到对象中对应的属性并直接调用。当函数作为方法调用时，函数体内叫做 `this` 的绑定自动指向在它上面调用的对象。

```
function speak(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
}
let whiteRabbit = {type: "white", speak: speak};
let fatRabbit = {type: "fat", speak: speak};

whiteRabbit.speak("Oh my ears and whiskers, " +
                  "how late it's getting!");
// → The white rabbit says 'Oh my ears and whiskers, how
//   late it's getting!'
hungryRabbit.speak("I could use a carrot right now.");
// → The hungry rabbit says 'I could use a carrot right now.'
```

你可以把 `this` 看作是以不同方式传递的额外参数。 如果你想显式传递它，你可以使用函数的 `call` 方法，它接受 `this` 值作为第一个参数，并将其它处理为看做普通参数。

```
speak.call(hungryRabbit, "Burp!");
// → The hungry rabbit says 'Burp!'
```

这段代码使用了关键字 `this` 来输出正在说话的兔子的种类。我们回想一下 `apply` 和 `bind` 方法，这两个方法接受的第一个参数可以用来模拟对象中方法的调用。这两个方法会把第一个参数复制给 `this` 。

由于每个函数都有自己的 `this` 绑定，它的值依赖于它的调用方式，所以在用 `function` 关键字定义的常规函数中，不能引用外层作用域的 `this` 。

箭头函数是不同的 - 它们不绑定他们自己的 `this` ，但可以看到他们周围（定义位置）作用域的 `this` 绑定。 因此，你可以像下面的代码那样，在局部函数中引用 `this` ：

```
function normalize() {
  console.log(this.coords.map(n => n / this.length));
}
normalize.call({coords: [0, 2, 3], length: 5});
// → [0, 0.4, 0.6]
```

如果我使用 `function` 关键字将参数写入 `map` ，则代码将不起作用。

原型

我们来仔细看看以下这段代码。

```
let empty = {};
console.log(empty.toString);
// → function toString(){…}
console.log(empty.toString());
// → [object Object]
```

我从一个空对象中取出了一个属性。 好神奇！

实际上并非如此。我只是掩盖了一些 JavaScript 对象的内部工作细节罢了。每个对象除了拥有自己的属性外，都包含一个原型（prototype）。原型是另一个对象，是对象的一个属性来源。当开发人员访问一个对象不包含的属性时，就会从对象原型中搜索属性，接着是原型的原型，依此类推。

那么空对象的原型是什么呢？是 `Object.prototype` ，它是所有对象中原型的父原型。

```
console.log(Object.getPrototypeOf({}) ==
            Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

正如你的猜测， `Object.getPrototypeOf` 返回一个对象的原型。

JavaScript 对象原型的关系是一种树形结构，整个树形结构的根部就是 `Object.prototype` 。`Object.prototype` 提供了一些可以在所有对象中使用的方法。比如说，`toString` 方法可以将一个对象转换成其字符串表示形式。

许多对象并不直接将 `Object.prototype` 作为其原型，而会使用另一个原型对象，用于提供一系列不同的默认属性。函数继承自 `Function.prototype` ，而数组继承自 `Array.prototype` 。

```
console.log(Object.getPrototypeOf(Math.max) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
            Array.prototype);
// → true
```

对于这样的原型对象来说，其自身也包含了一个原型对象，通常情况下是 `Object.prototype` ，所以说，这些原型对象可以间接提供 `toString` 这样的方法。

你可以使用 `Object.create` 来创建一个具有特定原型的对象。

```
let protoRabbit = {
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
};
let killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "killer";
killerRabbit.speak("SKREEEE!");
// → The killer rabbit says 'SKREEEE!'
```

像对象表达式中的 `speak(line)` 这样的属性是定义方法的简写。 它创建了一个名为 `speak` 的属性，并向其提供函数作为它的值。

原型对象 `protoRabbit` 是一个容器，用于包含所有兔子对象的公有属性。每个独立的兔子对象（比如 `killerRabbit` ）可以包含其自身属性（比如本例中的 `type` 属性），也可以派生其原型对象中公有的属性。

类

JavaScript 的原型系统可以解释为对一种面向对象的概念（称为类（class））的某种非正式实现。 类定义了对象的类型的形状 - 它具有什么方法和属性。 这样的对象被称为类的实例（instance）。

原型对于属性来说很实用。一个类的所有实例共享相同的属性值，例如方法。 每个实例上的不同属性，比如我们的兔子的 `type` 属性，需要直接存储在对象本身中。

所以为了创建一个给定类的实例，你必须使对象从正确的原型派生，但是你也必须确保，它本身具有这个类的实例应该具有的属性。 这是构造器（constructor）函数的作用。

```
function makeRabbit(type) {
  let rabbit = Object.create(protoRabbit);
  rabbit.type = type;
  return rabbit;
}
```

JavaScript 提供了一种方法，来使得更容易定义这种类型的功能。 如果将关键字 `new` 放在函数调用之前，则该函数将被视为构造器。 这意味着具有正确原型的对象会自动创建，绑定到函数中的 `this`，并在函数结束时返回。

构造对象时使用的原型对象，可以通过构造器的 `prototype` 属性来查找。

```
function Rabbit(type) {
  this.type = type;
}

Rabbit.prototype.speak = function(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
};
let weirdRabbit = new Rabbit("weird");
```

构造器（实际上是所有函数）都会自动获得一个名为 `prototype` 的属性，默认情况下它包含一个普通的，来自 `Object.prototype` 的空对象。 如果需要，可以用新对象覆盖它。 或者，你可以将属性添加到现有对象，如示例所示。

按照惯例，构造器的名字是大写的，这样它们可以很容易地与其他函数区分开来。

重要的是，理解原型与构造器关联的方式（通过其 `prototype` 属性），与对象拥有原型（可以通过 `Object.getPrototypeOf` 查找）的方式之间的区别。 构造器的实际原型是 `Function.prototype`，因为构造器是函数。 它的 `prototype` 属性拥有原型，用于通过它创建的实例。

```
console.log(Object.getPrototypeOf(Rabbit) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf(weirdRabbit) ==
            Rabbit.prototype);
// → true
```

类的表示法

所以 JavaScript 类是带有原型属性的构造器。 这就是他们的工作方式，直到 2015 年，这就是你编写他们的方式。 最近，我们有了一个不太笨拙的表示法。

```
class Rabbit {
  constructor(type) {
    this.type = type;
  }
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
}

let killerRabbit = new Rabbit("killer");
let blackRabbit = new Rabbit("black");
```

`class` 关键字是类声明的开始，它允许我们在一个地方定义一个构造器和一组方法。 可以在声明的大括号内写入任意数量的方法。 一个名为 `constructor` 的对象受到特别处理。 它提供了实际的构造器，它将绑定到名称 `"Rabbit"` 。 其他函数被打包到该构造器的原型中。 因此，上面的类声明等同于上一节中的构造器定义。 它看起来更好。

类声明目前只允许方法 - 持有函数的属性 - 添加到原型中。 当你想在那里保存一个非函数值时，这可能会有点不方便。 该语言的下一个版本可能会改善这一点。 现在，你可以在定义该类后直接操作原型来创建这些属性。

像 `function` 一样， `class` 可以在语句和表达式中使用。 当用作表达式时，它没有定义绑定，而只是将构造器作为一个值生成。 你可以在类表达式中省略类名称。

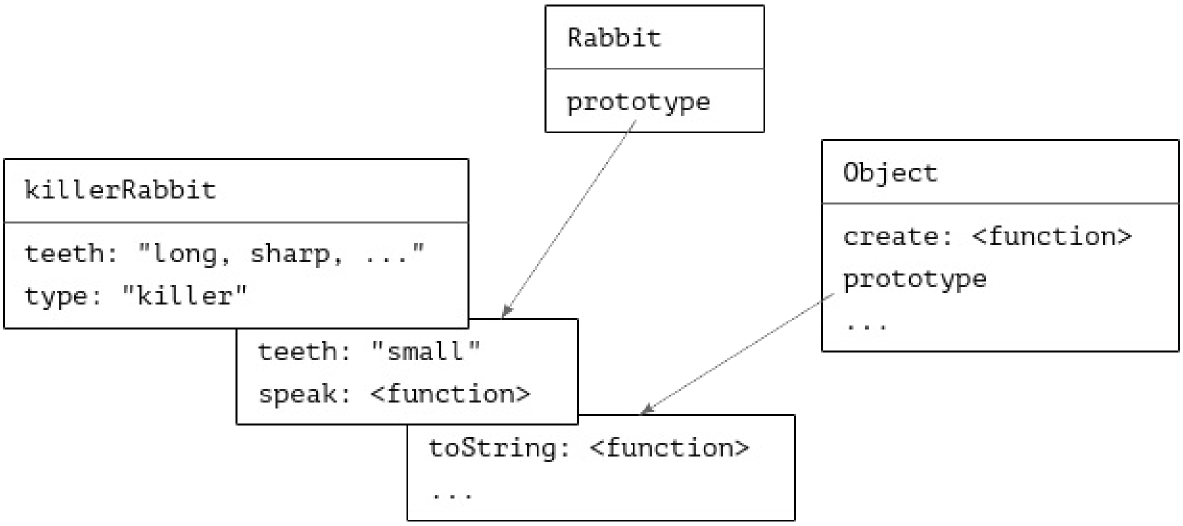
```
let object = new class { getWord() { return "hello"; } };
console.log(object.getWord());
// → hello
```

覆盖派生的属性

将属性添加到对象时，无论它是否存在于原型中，该属性都会添加到对象本身中。 如果原型中已经有一个同名的属性，该属性将不再影响对象，因为它现在隐藏在对象自己的属性后面。

```
Rabbit.prototype.teeth = "small";
console.log(killerRabbit.teeth);
// → small
killerRabbit.teeth = "long, sharp, and bloody";
console.log(killerRabbit.teeth);
// → long, sharp, and bloody
console.log(blackRabbit.teeth);
// → small
console.log(Rabbit.prototype.teeth);
// → small
```

下图简单地描述了代码执行后的情况。其中 `Rabbit` 和 `Object` 原型画在了 `killerRabbit` 之下，我们可以从原型中找到对象中没有的属性。



覆盖原型中存在的属性是很有用的特性。就像示例展示的那样，我们覆盖了`killerRabbit`的`teeth`属性，这可以用来描述实例（对象中更为泛化的类的实例）的特殊属性，同时又可以让简单对象从原型中获取标准的值。

覆盖也用于向标准函数和数组原型提供`toString`方法，与基本对象的原型不同。

```
console.log(Array.prototype.toString ==
            Object.prototype.toString);
// → false
console.log([1, 2].toString());
// → 1,2
```

调用数组的`toString`方法后得到的结果与调用`.join(",")`的结果十分类似，即在数组的每个值之间插入一个逗号。而直接使用数组调用`Object.prototype.toString`则会产生一个完全不同的字符串。由于`Object`原型提供的`toString`方法并不了解数组结构，因此只会简单地输出一对方括号，并在方括号中间输出单词`"object"`和类型的名称。

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

映射

我们在上一章中看到了映射（map）这个词，用于一个操作，通过对元素应用函数来转换数据结构。 令人困惑的是，在编程时，同一个词也被用于相关而不同的事物。

映射（名词）是将值（键）与其他值相关联的数据结构。 例如，你可能想要将姓名映射到年龄。 为此可以使用对象。

```
let ages = {
  Boris: 39,
  Liang: 22,
  Júlia: 62
};

console.log(`Júlia is ${ages["Júlia"]}`);
// → Júlia is 62
console.log("Is Jack's age known?", "Jack" in ages);
// → Is Jack's age known? false
console.log("Is toString's age known?", "toString" in ages);
// → Is toString's age known? true
```

在这里，对象的属性名称是人们的姓名，并且该属性的值为他们的年龄。但是我们当然没有在我们的映射中列出任何名为 `toString` 的人。似的，因为简单对象是从 `Object.prototype` 派生的，所以它看起来就像拥有这个属性。

因此，使用简单对象作为映射是危险的。有几种可能的方法来避免这个问题。首先，可以使用 `null` 原型创建对象。如果将 `null` 传递给 `Object.create`，那么所得到的对象将不会从 `Object.prototype` 派生，并且可以安全地用作映射。

```
console.log("toString" in Object.create(null));
// → false
```

对象属性名称必须是字符串。如果你需要一个映射，它的键不能轻易转换为字符串 - 比如对象 - 你不能使用对象作为你的映射。

幸运的是，JavaScript 带有一个叫做 `Map` 的类，它正是为了这个目的而编写。它存储映射并允许任何类型的键。

```
let ages = new Map();
ages.set("Boris", 39);
ages.set("Liang", 22);
ages.set("Júlia", 62);
console.log(`Júlia is ${ages.get("Júlia")}`);
// → Júlia is 62
console.log("Is Jack's age known?", ages.has("Jack"));
// → Is Jack's age known? false
console.log(ages.has("toString"));
 // → false
```

`set`，`get` 和 `has` 方法是 `Map` 对象的接口的一部分。编写一个可以快速更新和搜索大量值的数据结构并不容易，但我们不必担心这一点。其他人为我们实现，我们可以通过这个简单的接口来使用他们的工作。

如果你确实有一个简单对象，出于某种原因需要将它视为一个映射，那么了解 `Object.keys` 只返回对象的自己的键，而不是原型中的那些键，会很有用。作为 `in` 运算符的替代方法，你可以使用 `hasOwnProperty` 方法，该方法会忽略对象的原型。

```
console.log({x: 1}.hasOwnProperty("x"));
// → true
console.log({x: 1}.hasOwnProperty("toString"));
// → false
```

多态

当你调用一个对象的`String`函数（将一个值转换为一个字符串）时，它会调用该对象的`toString`方法来尝试从它创建一个有意义的字符串。我提到一些标准原型定义了自己的`toString`版本，因此它们可以创建一个包含比`"[object Object]"`有用信息更多的字符串。你也可以自己实现。

```
Rabbit.prototype.toString = function() {
  return `a ${this.type} rabbit`;
};

console.log(String(blackRabbit));
// → a black rabbit
```

这是一个强大的想法的简单实例。当一段代码为了与某些对象协作而编写，这些对象具有特定接口时（在本例中为`toString`方法），任何类型的支持此接口的对象都可以插入到代码中，并且它将正常工作。

这种技术被称为多态（polymorphism）。多态代码可以处理不同形状的值，只要它们支持它所期望的接口即可。

我在第四章中提到`for/of`循环可以遍历几种数据结构。这是多态性的另一种情况 - 这样的循环期望数据结构公开的特定接口，数组和字符串是这样。你也可以将这个接口添加到你自己的对象中！但在我们实现它之前，我们需要知道什么是符号。

符号

多个接口可能为不同的事物使用相同的属性名称。例如，我可以定义一个接口，其中`toString`方法应该将对象转换为一段纱线。一个对象不可能同时满足这个接口和`toString`的标准用法。

这是一个坏主意，这个问题并不常见。大多数 JavaScript 程序员根本就不会去想它。但是，语言设计师们正在思考这个问题，无论如何都为我们提供了解决方案。

当我声称属性名称是字符串时，这并不完全准确。他们通常是，但他们也可以是符号（symbol）。符号是使用`Symbol`函数创建的值。与字符串不同，新创建的符号是唯一的 - 你不能两次创建相同的符号。

```
let sym = Symbol("name");
console.log(sym == Symbol("name"));
// → false
Rabbit.prototype[sym] = 55;
console.log(blackRabbit[sym]);
// → 55
```

将`Symbol`转换为字符串时，会得到传递给它的字符串，例如，在控制台中显示时，符号可以更容易识别。 但除此之外没有任何意义 - 多个符号可能具有相同的名称。

由于符号既独特又可用于属性名称，因此符号适合定义可以和其他属性共生的接口，无论它们的名称是什么。

```
const toStringSymbol = Symbol("toString");
Array.prototype[toStringSymbol] = function() {
  return `${this.length} cm of blue yarn`;
};
console.log([1, 2].toString());
// → 1,2
console.log([1, 2][toStringSymbol]());
// → 2 cm of blue yarn
```

通过在属性名称周围使用方括号，可以在对象表达式和类中包含符号属性。 这会导致属性名称的求值，就像方括号属性访问表示法一样，这允许我们引用一个持有该符号的绑定。

```
let stringObject = {
  [toStringSymbol]() { return "a jute rope"; }
};
console.log(stringObject[toStringSymbol]());
// → a jute rope
```

迭代器接口

提供给`for/of`循环的对象预计为可迭代对象（iterable）。 这意味着它有一个以`Symbol.iterator`符号命名的方法（由语言定义的符号值，存储为`Symbol`符号的一个属性）。

当被调用时，该方法应该返回一个对象，它提供第二个接口迭代器（iterator）。 这是执行迭代的实际事物。 它拥有返回下一个结果的`next`方法。 这个结果应该是一个对象，如果有下一个值，`value`属性会提供它；没有更多结果时，`done`属性应该为`true`，否则为`false`。

请注意，`next`，`value`和`done`属性名称是纯字符串，而不是符号。 只有`Symbol.iterator`是一个实际的符号，它可能被添加到不同的大量对象中。

我们可以直接使用这个接口。

```
let okIterator = "OK"[Symbol.iterator]();
console.log(okIterator.next());
// → {value: "O", done: false}
console.log(okIterator.next());
// → {value: "K", done: false}
console.log(okIterator.next());
// → {value: undefined, done: true}
```

我们来实现一个可迭代的数据结构。 我们将构建一个 `matrix` 类，充当一个二维数组。

```
class Matrix {
  constructor(width, height, element = (x, y) => undefined) {
    this.width = width;
    this.height = height;
    this.content = [];
    for (let y = 0; y < height; y++) {
      for (let x = 0; x < width; x++) {
        this.content[y * width + x] = element(x, y);
      }
    }
  }
  get(x, y) {
    return this.content[y * this.width + x];
  }
  set(x, y, value) {
    this.content[y * this.width + x] = value;
  }
}
```

该类将其内容存储在 `width × height` 个元素的单个数组中。 元素是按行存储的，因此，例如，第五行中的第三个元素存储在位置 `4 × width + 2` 中（使用基于零的索引）。

构造器需要宽度，高度和一个可选的内容函数，用来填充初始值。 `get` 和 `set` 方法用于检索和更新矩阵中的元素。

遍历矩阵时，通常对元素的位置以及元素本身感兴趣，所以我们会让迭代器产生具有 `x` ， `y` 和 `value` 属性的对象。

```
class MatrixIterator {
  constructor(matrix) {
    this.x = 0;
    this.y = 0;
    this.matrix = matrix;
  }
  next() {
    if (this.y == this.matrix.height) return {done: true};

    let value = {x: this.x,
                 y: this.y,
                 value: this.matrix.get(this.x, this.y)};
    this.x++;
    if (this.x == this.matrix.width) {
      this.x = 0;
      this.y++;
    }
    return {value, done: false};
  }
}
```

这个类在其 `x` 和 `y` 属性中跟踪遍历矩阵的进度。 `next` 方法最开始检查是否到达矩阵的底部。 如果没有，则首先创建保存当前值的对象，之后更新其位置，如有必要则移至下一行。

让我们使 `Matrix` 类可迭代。 在本书中，我会偶尔使用事后的原型操作来为类添加方法，以便单个代码段保持较小且独立。 在一个正常的程序中，不需要将代码分成小块，而是直接在 `class` 中声明这些方法。

```
Matrix.prototype[Symbol.iterator] = function() {
  return new MatrixIterator(this);
};
```

现在我们可以用 `for/of` 来遍历一个矩阵。

```
let matrix = new Matrix(2, 2, (x, y) => `value ${x},${y}`);
for (let {x, y, value} of matrix) {
  console.log(x, y, value);
}
// → 0 0 value 0,0
// → 1 0 value 1,0
// → 0 1 value 0,1
// → 1 1 value 1,1
```

读写器和静态

接口通常主要由方法组成，但也可以持有非函数值的属性。 例如， `Map` 对象有 `size` 属性，告诉你有多少个键存储在它们中。

这样的对象甚至不需要直接在实例中计算和存储这样的属性。 即使直接访问的属性也可能隐藏了方法调用。 这种方法称为读取器（getter），它们通过在方法名称前面编写 `get` 来定义。

```
let varyingSize = {
  get size() {
    return Math.floor(Math.random() * 100);
  }
};
console.log(varyingSize.size);
// → 73
console.log(varyingSize.size);
// → 49
```

每当有人读取此对象的 `size` 属性时，就会调用相关的方法。 当使用写入器（setter）写入属性时，可以做类似的事情。

```
class Temperature {
  constructor(celsius) {
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
  set fahrenheit(value) {
    this.celsius = (value - 32) / 1.8;
  }

  static fromFahrenheit(value) {
    return new Temperature((value - 32) / 1.8);
  }
}
let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

`Temperature`类允许你以摄氏度或华氏度读取和写入温度，但内部仅存储摄氏度，并在`fahrenheit`读写器中自动转换为摄氏度。

有时候你想直接向你的构造器附加一些属性，而不是原型。 这样的方法将无法访问类实例，但可以用来提供额外方法来创建实例。

在类声明内部，名称前面写有`static`的方法，存储在构造器中。 所以`Temperature`类可以让你写出`Temperature.fromFahrenheit(100)`，来使用华氏温度创建一个温度。

继承

已知一些矩阵是对称的。 如果沿左上角到右下角的对角线翻转对称矩阵，它保持不变。 换句话说，存储在`x,y`的值总是与`y,x`相同。

想象一下，我们需要一个像`Matrix`这样的数据结构，但是它必需保证一个事实，矩阵是对称的。 我们可以从头开始编写它，但这需要重复一些代码，与我们已经写过的代码很相似。

JavaScript 的原型系统可以创建一个新类，就像旧类一样，但是它的一些属性有了新的定义。 新类派生自旧类的原型，但为`set`方法增加了一个新的定义。

在面向对象的编程术语中，这称为继承（inheritance）。 新类继承旧类的属性和行为。

```
class SymmetricMatrix extends Matrix {
  constructor(size, element = (x, y) => undefined) {
    super(size, size, (x, y) => {
      if (x < y) return element(y, x);
      else return element(x, y);
    });
  }

  set(x, y, value) {
    super.set(x, y, value);
    if (x != y) {
      super.set(y, x, value);
    }
  }
}
let matrix = new SymmetricMatrix(5, (x, y) => `${x},${y}`);
console.log(matrix.get(2, 3));
// → 3,2
```

`extends`这个词用于表示，这个类不应该直接基于默认的`Object`原型，而应该基于其他类。这被称为超类（superclass）。派生类是子类（subclass）。

为了初始化`SymmetricMatrix`实例，构造器通过`super`关键字调用其超类的构造器。这是必要的，因为如果这个新对象的行为（大致）像`Matrix`，它需要矩阵具有的实例属性。为了确保矩阵是对称的，构造器包装了`content`方法，来交换对角线以下的值的坐标。

`set`方法再次使用`super`，但这次不是调用构造器，而是从超类的一组方法中调用特定的方法。我们正在重新定义`set`，但是想要使用原来的行为。因为`this.set`引用新的`set`方法，所以调用这个方法是行不通的。在类方法内部，`super`提供了一种方法，来调用超类中定义的方法。

继承允许我们用相对较少的工作，从现有数据类型构建稍微不同的数据类型。它是面向对象传统的基础部分，与封装和多态一样。尽管后两者现在普遍被认为是伟大的想法，但继承更具争议性。

尽管封装和多态可用于将代码彼此分离，从而减少整个程序的耦合，但继承从根本上将类连接在一起，从而产生更多的耦合。继承一个类时，比起单纯使用它，你通常必须更加了解它如何工作。继承可能是一个有用的工具，并且我现在在自己的程序中使用它，但它不应该成为你的第一个工具，你可能不应该积极寻找机会来构建类层次结构（类的家族树）。

`instanceof`运算符

在有些时候，了解某个对象是否继承自某个特定类，也是十分有用的。JavaScript 为此提供了一个二元运算符，名为`instanceof`。

```
console.log(
  new SymmetricMatrix(2) instanceof SymmetricMatrix);
// → true
console.log(new SymmetricMatrix(2) instanceof Matrix);
// → true
console.log(new Matrix(2, 2) instanceof SymmetricMatrix);
// → false
console.log([1] instanceof Array);
// → true
```

该运算符会浏览所有继承类型。所以`SymmetricMatrix`是`Matrix`的一个实例。该运算符也可以应用于像`Array`这样的标准构造器。几乎每个对象都是`Object`的一个实例。

本章小结

对象不仅仅持有它们自己的属性。对象中有另一个对象：原型，只要原型中包含了属性，那么根据原型构造出来的对象也就可以看成包含了相应的属性。简单对象直接以`Object.prototype`作为原型。

构造器是名称通常以大写字母开头的函数，可以与`new`运算符一起使用来创建新对象。新对象的原型是构造器的`prototype`属性中的对象。通过将属性放到它们的原型中，可以充分利用这一点，给定类型的所有值在原型中分享它们的属性。`class`表示法提供了一个显式方法，来定义一个构造器及其原型。

你可以定义读写器，在每次访问对象的属性时秘密地调用方法。静态方法是存储在类的构造器，而不是其原型中的方法。

给定一个对象和一个构造器，`instanceof`运算符可以告诉你该对象是否是该构造器的一个实例。

可以使用对象的来做一个有用的事情是，为它们指定一个接口，告诉每个人他们只能通过该接口与对象通信。构成对象的其余细节，现在被封装在接口后面。

不止一种类型可以实现相同的接口。为使用接口而编写的代码，自动知道如何使用提供接口的任意数量的不同对象。这被称为多态。

实现多个类，它们仅在一些细节上有所不同的时，将新类编写为现有类的子类，继承其一部分行为会很有帮助。

习题

向量类型

编写一个构造器`Vec`，在二维空间中表示数组。该函数接受两个数字参数`x`和`y`，并将其保存到对象的同名属性中。

向 `Vec` 原型添加两个方法：`plus` 和 `minus`，它们接受另一个向量作为参数，分别返回两个向量（一个是 `this`，另一个是参数）的和向量与差向量。

向原型添加一个 `getter` 属性 `length`，用于计算向量长度，即点 `(x,y)` 与原点 `(0,0)` 之间的距离。

```
// Your code here.

console.log(new Vec(1, 2).plus(new Vec(2, 3)));
// → Vec{x: 3, y: 5}
console.log(new Vec(1, 2).minus(new Vec(2, 3)));
// → Vec{x: -1, y: -1}
console.log(new Vec(3, 4).length);
// → 5
```

分组

标准的 JavaScript 环境提供了另一个名为 `Set` 的数据结构。 像 `Map` 的实例一样，集合包含一组值。 与 `Map` 不同，它不会将其他值与这些值相关联 - 它只会跟踪哪些值是该集合的一部分。 一个值只能是一个集合的一部分 - 再次添加它没有任何作用。

写一个名为 `Group` 的类（因为 `Set` 已被占用）。 像 `Set` 一样，它具有 `add`，`delete` 和 `has` 方法。 它的构造器创建一个空的分组，`add` 给分组添加一个值（但仅当它不是成员时），`delete` 从组中删除它的参数（如果它是成员），`has` 返回一个布尔值，表明其参数是否为分组的成员。

使用 `===` 运算符或类似于 `indexOf` 的东西来确定两个值是否相同。

为该类提供一个静态的 `from` 方法，该方法接受一个可迭代的对象作为参数，并创建一个分组，包含遍历它产生的所有值。

```
// Your code here.

class Group {
  // Your code here.
}
let group = Group.from([10, 20]);
console.log(group.has(10));
// → true
console.log(group.has(30));
// → false
group.add(10);
group.delete(10);
console.log(group.has(10));
// → false
```

可迭代分组

使上一个练习中的 `Group` 类可迭代。 如果你不清楚接口的确切形式，请参阅本章前面迭代器接口的章节。

如果你使用数组来表示分组的成员，则不要仅仅通过调用数组中的`Symbol.iterator`方法来返回迭代器。 这会起作用，但它会破坏这个练习的目的。

如果分组被修改时，你的迭代器在迭代过程中出现奇怪的行为，那也没问题。

```
// Your code here (and the code from the previous exercise)

for (let value of Group.from(["a", "b", "c"])) {
  console.log(value);
}
// → a
// → b
// → c
```

借鉴方法

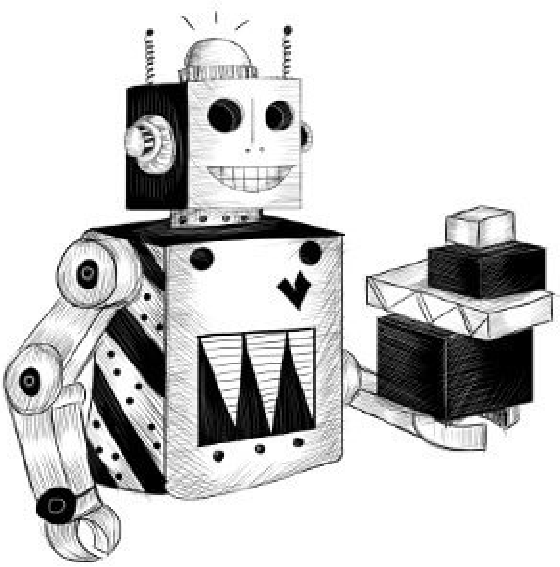
在本章前面我提到，当你想忽略原型的属性时，对象的`hasOwnProperty`可以用作`in`运算符的更强大的替代方法。 但是如果你的映射需要包含`hasOwnProperty`这个词呢？ 你将无法再调用该方法，因为对象的属性隐藏了方法值。

你能想到一种方法，对拥有自己的同名属性的对象，调用`hasOwnProperty`吗？

```
let map = {one: true, two: true, hasOwnProperty: true};
// Fix this call
console.log(map.hasOwnProperty("one"));
// → true
```

七、项目：机器人

> 原文：Project: A Robot
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> [...] 置疑计算机能不能思考 [...] 就相当于置疑潜艇能不能游泳。
>
> 艾兹格尔·迪科斯特拉，《计算机科学的威胁》



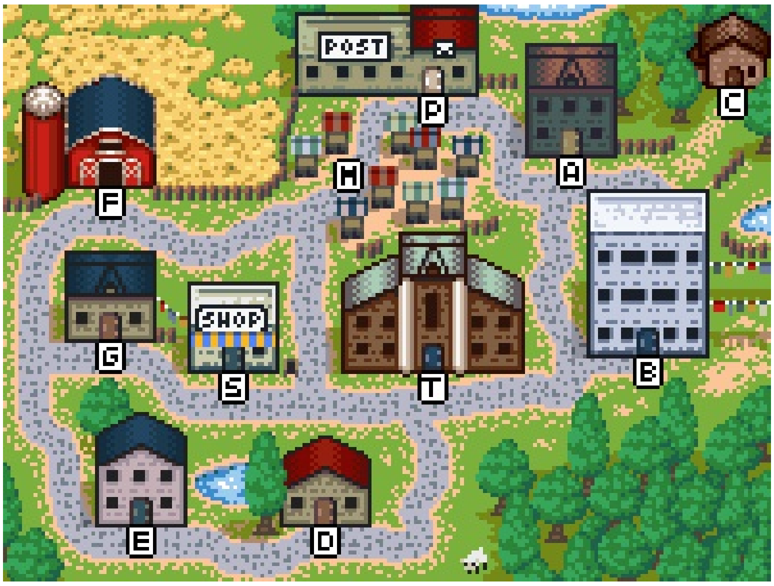
在“项目”章节中，我会在短时间内停止向你讲述新理论，相反我们会一起完成一个项目。 学习编程理论是必要的，但阅读和理解实际的计划同样重要。

我们在本章中的项目是构建一个自动机，一个在虚拟世界中执行任务的小程序。 我们的自动机将是一个接送包裹的邮件递送机器人。

Meadowfield

Meadowfield 村不是很大。 它由 11 个地点和 14 条道路组成。 它可以用 `roads` 数组来描述：

```
const roads = [
  "Alice's House-Bob's House",   "Alice's House-Cabin",
  "Alice's House-Post Office",   "Bob's House-Town Hall",
  "Daria's House-Ernie's House", "Daria's House-Town Hall",
  "Ernie's House-Grete's House", "Grete's House-Farm",
  "Grete's House-Shop",          "Marketplace-Farm",
  "Marketplace-Post Office",     "Marketplace-Shop",
  "Marketplace-Town Hall",       "Shop-Town Hall"
];
```



村里的道路网络形成了一个图。 图是节点（村里的地点）与他们之间的边（道路）的集合。这张图将成为我们的机器人在其中移动的世界。

字符串数组并不易于处理。 我们感兴趣的是，我们可以从特定地点到达的目的地。 让我们将道路列表转换为一个数据结构，对于每个地点，都会告诉我们从那里可以到达哪些地点。

```
function buildGraph(edges) {
  let graph = Object.create(null);
  function addEdge(from, to) {
    if (graph[from] == null) {
      graph[from] = [to];
    } else {
      graph[from].push(to);
    }
  }
  for (let [from, to] of edges.map(r => r.split("-"))) {
    addEdge(from, to);
    addEdge(to, from);
  }
  return graph;
}

const roadGraph = buildGraph(roads);
```

给定边的数组，`buildGraph`创建一个映射对象，该对象为每个节点存储连通节点的数组。

它使用`split`方法，将形式为`"Start-End"`的道路字符串，转换为两元素数组，包含起点和终点作为单个字符串。

任务

我们的机器人将在村庄周围移动。 在各个地方都有包裹，每个都寄往其他地方。 机器人在收到包裹时拾取包裹，并在抵达目的地时将其送达。

自动机必须在每个点决定下一步要去哪里。 所有包裹递送完成后，它就完成了任务。

为了能够模拟这个过程，我们必须定义一个可以描述它的虚拟世界。 这个模型告诉我们机器人在哪里以及包裹在哪里。 当机器人决定移到某处时，我们需要更新模型以反映新情况。

如果你正在考虑面向对象编程，你的第一个冲动可能是开始为世界中的各种元素定义对象。一个机器人，一个包裹，也许还有一个地点。 然后，它们可以持有描述其当前状态的属性，例如某个位置的一堆包裹，我们可以在更新世界时改变这些属性。

这是错的。

至少，通常是这样。 一个东西听起来像一个对象，并不意味着它应该是你的程序中的一个对象。 为应用程序中的每个概念反射式编写类，往往会留下一系列互连对象，每个对象都有自己的内部的变化的状态。 这样的程序通常很难理解，因此很容易崩溃。

相反，让我们将村庄的状态压缩成定义它的值的最小集合。 机器人的当前位置和未送达的包裹集合，其中每个都拥有当前位置和目标地址。这样就够了。

当我们到达新地点时，让我们这样做，在机器人移动时不会改变这种状态，而是在移动之后为当前情况计算一个新状态。

```
class VillageState {
  constructor(place, parcels) {
    this.place = place;
    this.parcels = parcels;
  }

  move(destination) {
    if (!roadGraph[this.place].includes(destination)) {
      return this;
    } else {
      let parcels = this.parcels.map(p => {
        if (p.place != this.place) return p;
        return {place: destination, address: p.address};
      }).filter(p => p.place != p.address);
      return new VillageState(destination, parcels);
    }
  }
}
```

`move` 方法是动作发生的地方。 它首先检查是否有当前位置到目的地的道路，如果没有，则返回旧状态，因为这不是有效的移动。

然后它创建一个新的状态，将目的地作为机器人的新地点。 但它也需要创建一套新的包裹 - 机器人携带的包裹（位于机器人当前位置）需要移动到新位置。 而要寄往新地点的包裹需要送达 - 也就是说，需要将它们从未送达的包裹中移除。 `'map'` 的调用处理移动，并

且 `'filter'` 的调用处理递送。

包裹对象在移动时不会更改，但会被重新创建。 `move` 方法为我们提供新的村庄状态，但完全保留了原有的村庄状态。

```
let first = new VillageState(
  "Post Office",
  [{place: "Post Office", address: "Alice's House"}]
);
let next = first.move("Alice's House");

console.log(next.place);
// → Alice's House
console.log(next.parcels);
// → []
console.log(first.place);
// → Post Office
```

`move` 会使包裹被送达，并在下一个状态中反映出来。 但最初的状态仍然描述机器人在邮局并且包裹未送达的情况。

持久性数据

不会改变的数据结构称为不变的（immutable）或持久性的（persistent）。 他们的表现很像字符串和数字，因为他们就是他们自己，并保持这种状态，而不是在不同的时间包含不同的东西。

在 JavaScript 中，几乎所有的东西都可以改变，所以使用应该持久性的值需要一些限制。 有一个叫做 `Object.freeze` 的函数，它可以改变一个对象，使其忽略它的属性的写入。 如果你想要小心，你可以使用它来确保你的对象没有改变。 `freeze` 确实需要计算机做一些额外的工作，忽略更新可能会让一些人迷惑，让他们做错事。 所以我通常更喜欢告诉人们，不应该弄乱给定的对象，并希望他们记住它。

```
let object = Object.freeze({value: 5});
object.value = 10;
console.log(object.value);
// → 5
```

当语言显然期待我这样做时，为什么我不想改变对象？

因为它帮助我理解我的程序。 这又是关于复杂性管理。 当我的系统中的对象是固定的，稳定的东西时，我可以孤立地考虑操作它们 - 从给定的起始状态移动到爱丽丝的房子，始终会产生相同的新状态。 当对象随着时间而改变时，这就给这种推理增加了全新的复杂性。

对于小型系统，例如我们在本章中构建的东西，我们可以处理那些额外的复杂性。 但是我们可以建立什么样的系统，最重要的限制是我们能够理解多少。 任何让你的代码更容易理解的东西，都可以构建一个更加庞大的系统。

不幸的是，尽管理解构建在持久性数据结构上的系统比较容易，但设计一个，特别是当你的编程语言没有帮助时，可能会更难一些。 我们将在本书中寻找使用持久性数据结构的时机，但我们也将使用可变数据结构。

模拟

递送机器人观察世界并决定它想要移动的方向。 因此，我们可以说机器人是一个函数，接受 `VillageState` 对象并返回附近地点的名称。

因为我们希望机器人能够记住东西，以便他们可以制定和执行计划，我们也会传递他们的记忆，并让他们返回一个新的记忆。 因此，机器人返回的东西是一个对象，包含它想要移动的方向，以及下次调用时将返回给它的记忆值。

```
function runRobot(state, robot, memory) {
  for (let turn = 0;; turn++) {
    if (state.parcels.length == 0) {
      console.log(`Done in ${turn} turns`);
      break;
    }
    let action = robot(state, memory);
    state = state.move(action.direction);
    memory = action.memory;
    console.log(`Moved to ${action.direction}`);
  }
}
```

考虑一下机器人必须做些什么来“解决”一个给定的状态。 它必须通过访问拥有包裹的每个位置来拾取所有包裹，并通过访问包裹寄往的每个位置来递送，但只能在拾取包裹之后。

什么是可能有效的最愚蠢的策略？ 机器人可以在每回合中，向随机方向行走。 这意味着很有可能它最终会碰到所有的包裹，然后也会在某个时候到达包裹应该送达的地方。

以下是可能的样子：

```
function randomPick(array) {
  let choice = Math.floor(Math.random() * array.length);
  return array[choice];
}

function randomRobot(state) {
  return {direction: randomPick(roadGraph[state.place])};
}
```

请记住，`Math.random()` 返回 0 和 1 之间的数字，但总是小于 1。 将这样一个数乘以数组长度，然后将 `Math.floor` 应用于它，向我们提供数组的随机索引。

由于这个机器人不需要记住任何东西，所以它忽略了它的第二个参数（记住，可以使用额外的参数调用 JavaScript 函数而不会产生不良影响）并省略返回对象中的 `memory` 属性。

为了使这个复杂的机器人工作，我们首先需要一种方法来创建一些包裹的新状态。 静态方法（通过直接向构造函数添加一个属性来编写）是放置该功能的好地方。

```
VillageState.random = function(parcelCount = 5) {
  let parcels = [];
  for (let i = 0; i < parcelCount; i++) {
    let address = randomPick(Object.keys(roadGraph));
    let place;
    do {
      place = randomPick(Object.keys(roadGraph));
    } while (place == address);
    parcels.push({place, address});
  }
  return new VillageState("Post Office", parcels);
};
```

我们不想要发往寄出地的任何包裹。 出于这个原因，当`do`循环获取与地址相同的地方时，它会继续选择新的地方。

让我们建立一个虚拟世界。

```
runRobot(VillageState.random(), randomRobot);
// → Moved to Marketplace
// → Moved to Town Hall
// → …
// → Done in 63 turns
```

机器人需要花费很多时间来交付包裹，因为它没有很好规划。 我们很快就会解决。

为了更好地理解模拟，你可以使用本章编程环境中提供的`runRobotAnimation`函数。 这将运行模拟，但不是输出文本，而是向你展示机器人在村庄地图上移动。

```
runRobotAnimation(VillageState.random(), randomRobot);
```

`runRobotAnimation`的实现方式现在仍然是一个谜，但是在阅读本书的后面的章节，讨论 Web 浏览器中的 JavaScript 集成之后，你将能够猜到它的工作原理。

邮车的路线

我们应该能够比随机机器人做得更好。 一个简单的改进就是从现实世界的邮件传递方式中获得提示。 如果我们发现一条经过村庄所有地点的路线，机器人可以通行该路线两次，此时它保证能够完成。 这是一条这样的路线（从邮局开始）。

```
const mailRoute = [
  "Alice's House", "Cabin", "Alice's House", "Bob's House",
  "Town Hall", "Daria's House", "Ernie's House",
  "Grete's House", "Shop", "Grete's House", "Farm",
  "Marketplace", "Post Office"
];
```

为了实现路线跟踪机器人，我们需要利用机器人的记忆。 机器人将其路线的其余部分保存在其记忆中，并且每回合丢弃第一个元素。

```
function routeRobot(state, memory) {
  if (memory.length == 0) {
    memory = mailRoute;
  }
  return {direction: memory[0], memory: memory.slice(1)};
}
```

这个机器人已经快了很多。 它最多需要 26 个回合（13 步的路线的两倍），但通常要少一些。

```
runRobotAnimation(VillageState.random(), routeRobot, []);
```

寻路

不过，我不会盲目遵循固定的智能寻路行为。 如果机器人为需要完成的实际工作调整行为，它可以更高效地工作。

为此，它必须能够有针对性地朝着给定的包裹移动，或者朝着包裹必须送达的地点。 尽管如此，即使目标距离我们不止一步，也需要某种寻路函数。

在图上寻找路线的问题是一个典型的搜索问题。 我们可以判断一个给定的解决方案（路线）是否是一个有效的解决方案，但我们不能像 2 + 2 这样，直接计算解决方案。 相反，我们必须不断创建潜在的解决方案，直到找到有效的解决方案。

图上的可能路线是无限的。 但是当搜索 `A` 到 `B` 的路线时，我们只关注从 `A` 起始的路线。 我们也不关心两次访问同一地点的路线 - 这绝对不是最有效的路线。 这样可以减少查找者必须考虑的路线数量。

事实上，我们最感兴趣的是最短路线。 所以我们要确保，查看较长路线之前，我们要查看较短的路线。 一个好的方法是，从起点使路线“生长”，探索尚未到达的每个可到达的地方，直到路线到达目标。 这样，我们只探索潜在的有趣路线，并找到到目标的最短路线（或最短路线之一，如果有多条路线）。

这是一个实现它的函数：

```
function findRoute(graph, from, to) {
  let work = [{at: from, route: []}];
  for (let i = 0; i < work.length; i++) {
    let {at, route} = work[i];
    for (let place of graph[at]) {
      if (place == to) return route.concat(place);
      if (!work.some(w => w.at == place)) {
        work.push({at: place, route: route.concat(place)});
      }
    }
  }
}
```

探索必须按照正确的顺序完成 - 首先到达的地方必须首先探索。 我们不能到达一个地方就立即探索，因为那样意味着，从那里到达的地方也会被立即探索，以此类推，尽管可能还有其他更短的路径尚未被探索。

因此，该函数保留一个工作列表。 这是一系列应该探索的地方，以及让我们到那里的路线。它最开始只有起始位置和空路线。

然后，通过获取列表中的下一个项目并进行探索，来执行搜索，这意味着，会查看从该地点起始的所有道路。 如果其中之一是目标，则可以返回完成的路线。 否则，如果我们以前没有看过这个地方，就会在列表中添加一个新项目。 如果我们之前看过它，因为我们首先查看了短路线，我们发现，到达那个地方的路线较长，或者与现有路线一样长，我们不需要探索它。

你可以在视觉上将它想象成一个已知路线的网，从起始位置爬出来，在各个方向上均匀生长（但不会缠绕回去）。 只要第一条线到达目标位置，其它线就会退回起点，为我们提供路线。

我们的代码无法处理工作列表中没有更多工作项的情况，因为我们知道我们的图是连通的，这意味着可以从其他所有位置访问每个位置。 我们始终能够找到两点之间的路线，并且搜索不会失败。

```
function goalOrientedRobot({place, parcels}, route) {
  if (route.length == 0) {
    let parcel = parcels[0];
    if (parcel.place != place) {
      route = findRoute(roadGraph, place, parcel.place);
    } else {
      route = findRoute(roadGraph, place, parcel.address);
    }
  }
  return {direction: route[0], memory: route.slice(1)};
}
```

这个机器人使用它的记忆值作为移动方向的列表，就像寻路机器人一样。 无论什么时候这个列表是空的，它都必须弄清下一步该做什么。 它会取出集合中第一个未送达的包裹，如果该包裹还没有被拾取，则会绘制一条朝向它的路线。 如果包裹已经被拾取，它仍然需要送达，所以机器人会创建一个朝向递送地址的路线。

让我们看看如何实现。

```
runRobotAnimation(VillageState.random(),
                  goalOrientedRobot, []);
```

这个机器人通常在大约 16 个回合中，完成了送达 5 个包裹的任务。略好于 `routeRobot`，但仍然绝对不是最优的。

练习

测量机器人

很难通过让机器人解决一些场景来客观比较他们。也许一个机器人碰巧得到了更简单的任务，或者它擅长的那种任务，而另一个没有。

编写一个 `compareRobots`，接受两个机器人（和它们的起始记忆）。它应该生成 100 个任务，并让每个机器人解决每个这些任务。完成后，它应输出每个机器人每个任务的平均步数。

为了公平起见，请确保你将每个任务分配给两个机器人，而不是为每个机器人生成不同的任务。

```
function compareRobots(robot1, memory1, robot2, memory2) {
  // Your code here
}

compareRobots(routeRobot, [], goalOrientedRobot, []);
```

机器人的效率

你能写一个机器人，比 `goalOrientedRobot` 更快完成递送任务吗？如果你观察机器人的行为，它会做什么明显愚蠢的事情？如何改进它们？

如果你解决了上一个练习，你可能打算使用 `compareRobots` 函数来验证你是否改进了机器人。

```
// Your code here

runRobotAnimation(VillageState.random(), yourRobot, memory);
```

持久性分组

标准 JavaScript 环境中提供的大多数数据结构不太适合持久使用。数组有 `slice` 和 `concat` 方法，可以让我们轻松创建新的数组而不会损坏旧数组。但是 `Set` 没有添加或删除项目并创建新集合的方法。

编写一个新的类`PGroup`，类似于第六章中的`Group`类，它存储一组值。 像`Group`一样，它具有`add`，`delete`和`has`方法。

然而，它的`add`方法应该返回一个新的`PGroup`实例，并添加给定的成员，并保持旧的不变。与之类似，`delete`创建一个没有给定成员的新实例。

该类应该适用于任何类型的值，而不仅仅是字符串。 当与大量值一起使用时，它不一定非常高效。

构造函数不应该是类接口的一部分（尽管你绝对会打算在内部使用它）。 相反，有一个空的实例`PGroup.empty`，可用作起始值。

为什么只需要一个`PGroup.empty`值，而不是每次都创建一个新的空分组？

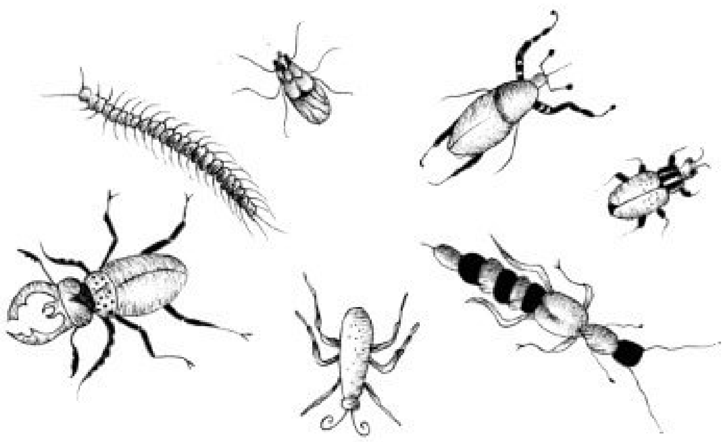
```
class PGroup {
  // Your code here
}

let a = PGroup.empty.add("a");
let ab = a.add("b");
let b = ab.delete("a");

console.log(b.has("b"));
// → true
console.log(a.has("b"));
// → false
console.log(b.has("a"));
// → false
```

八、Bug 和错误

> 原文：Bugs and Errors
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 调试的难度是开始编写代码的两倍。 因此，如果你尽可能巧妙地编写代码，那么根据定义，你的智慧不足以进行调试。
>
> Brian Kernighan 和 P.J. Plauger，《The Elements of Programming Style》



计算机程序中的缺陷通常称为 bug。 它让程序员觉得很好，将它们想象成小事，只是碰巧进入我们的作品。 实际上，当然，我们自己把它们放在了那里。

如果一个程序是思想的结晶，你可以粗略地将错误分为因为思想混乱引起的错误，以及思想转换为代码时引入的错误。 前者通常比后者更难诊断和修复。

语言

计算机能够自动地向我们指出许多错误，如果它足够了解我们正在尝试做什么。 但是这里 JavaScript 的宽松是一个障碍。 它的绑定和属性概念很模糊，在实际运行程序之前很少会发现拼写错误。 即使这样，它也允许你做一些不会报错的无意义的事情，比如计算 `true *'monkey'`。

JavaScript 有一些报错的事情。编写不符合语言语法的程序会立即使计算机报错。其他的东西，比如调用不是函数的东西，或者在未定义的值上查找属性，会导致在程序尝试执行操作时报告错误。

不过，JavaScript 在处理无意义的计算时，会仅仅返回 `NaN`（表示不是数字）或 `undefined` 这样的结果。程序会认为其执行的代码毫无问题并顺利运行下去，要等到随后的运行过程中才会出现问题，而此时已经有许多函数使用了这个无意义的值。程序执行中也可能不会遇到任何错误，只会产生错误的程序输出。找出这类错误的源头是非常困难的。

我们将查找程序中的错误或者 bug 的过程称为调试（debug）。

严格模式

当启用了严格模式（strict mode）后，JavaScript 就会在执行代码时变得更为严格。我们只需在文件或函数体顶部放置字符串 `"use strict"` 就可以启用严格模式了。下面是示例代码：

```
function canYouSpotTheProblem() {
  "use strict";
  for (counter = 0; counter < 10; counter++) {
    console.log("Happy happy");
  }
}

canYouSpotTheProblem();
// → ReferenceError: counter is not defined
```

通常，当你忘记在绑定前面放置 `let` 时，就像在示例中的 `counter` 一样，JavaScript 静静地创建一个全局绑定并使用它。在严格模式下，它会报告错误。这非常有帮助。但是，应该指出的是，当绑定已经作为全局绑定存在时，这是行不通的。在这种情况下，循环仍然会悄悄地覆盖绑定的值。

严格模式中的另一个变化是，在未被作为方法而调用的函数中，`this` 绑定持有值 `undefined`。当在严格模式之外进行这样的调用时，`this` 引用全局作用域对象，该对象的属性是全局绑定。因此，如果你在严格模式下不小心错误地调用方法或构造器，JavaScript 会在尝试从 `this` 读取某些内容时产生错误，而不是愉快地写入全局作用域。

例如，考虑下面的代码，该代码不带 `new` 关键字调用构造器，以便其 `this` 不会引用新构造的对象：

```
function Person(name) { this.name = name; }
let ferdinand = Person("Ferdinand"); // oops
console.log(name);
// → Ferdinand
```

虽然我们错误调用了 `Person`，代码也可以执行成功，但会返回一个未定义值，并创建名为 `name` 的全局绑定。而在严格模式中，结果就不同了。

```
"use strict";
function Person(name) { this.name = name; }
let ferdinand = Person("Ferdinand");
// → TypeError: Cannot set property 'name' of undefined
```

JavaScript 会立即告知我们代码中包含错误。这种特性十分有用。

幸运的是，使用`class`符号创建的构造器，如果在不使用`new`来调用，则始终会报错，即使在非严格模式下也不会产生问题。

严格模式做了更多的事情。它不允许使用同一名称给函数赋多个参数，并且完全删除某些有问题的语言特性（例如`with`语句，这是错误的，本书不会进一步讨论）。

简而言之，在程序顶部放置`"use strict"`很少会有问题，并且可能会帮助你发现问题。

类型

有些语言甚至在运行程序之前想要知道，所有绑定和表达式的类型。当类型以不一致的方式使用时，他们会马上告诉你。JavaScript 只在实际运行程序时考虑类型，即使经常尝试将值隐式转换为它预期的类型，所以它没有多大帮助。

尽管如此，类型为讨论程序提供了一个有用的框架。许多错误来自于值的类型的困惑，它们进入或来自一个函数。如果你把这些信息写下来，你不太可能会感到困惑。

你可以在上一章的`goalOrientedRobot`函数上面，添加一个像这样的注释来描述它的类型。

```
// (WorldState, Array) → {direction: string, memory: Array}
function goalOrientedRobot(state, memory) {
  // ...
}
```

有许多不同的约定，用于标注 JavaScript 程序的类型。

关于类型的一点是，他们需要引入自己的复杂性，以便能够描述足够有用的代码。你认为从数组中返回一个随机元素的`randomPick`函数的类型是什么？你需要引入一个绑定类型`T`，它可以代表任何类型，这样你就可以给予`randomPick`一个像`([T])->T`的类型（从`T`到`T`的数组的函数）。

当程序的类型已知时，计算机可以为你检查它们，在程序运行之前指出错误。有几种 JavaScript 语言为语言添加类型并检查它们。最流行的称为 TypeScript。如果你有兴趣为你的程序添加更多的严谨性，我建议你尝试一下。

在本书中，我们将继续使用原始的，危险的，非类型化的 JavaScript 代码。

测试

如果语言不会帮助我们发现错误，我们将不得不努力找到它们：通过运行程序并查看它是否正确执行。

一次又一次地手动操作，是一个非常糟糕的主意。 这不仅令人讨厌，而且也往往是无效的，因为每次改变时都需要花费太多时间来详尽地测试所有内容。

计算机擅长重复性任务，测试是理想的重复性任务。 自动化测试是编写测试另一个程序的程序的过程。 编写测试比手工测试有更多的工作，但是一旦你完成了它，你就会获得一种超能力：它只需要几秒钟就可以验证，你的程序在你编写为其测试的所有情况下都能正常运行。当你破坏某些东西时，你会立即注意到，而不是在稍后的时间里随机地碰到它。

测试通常采用小标签程序的形式来验证代码的某些方面。 例如，一组（标准的，可能已经由其他人测试过） `toUpperCase` 方法的测试可能如下：

```
function test(label, body) {
  if (!body()) console.log(`Failed: ${label}`);
}

test("convert Latin text to uppercase", () => {
  return "hello".toUpperCase() == "HELLO";
});
test("convert Greek text to uppercase", () => {
  return "Χαίρετε".toUpperCase() == "ΧΑΊΡΕΤΕ";
});
test("don't convert case-less characters", () => {
  return "مرحبا".toUpperCase() == "مرحبا";
});
```

像这样写测试往往会产生很多重复，笨拙的代码。 幸运的是，有些软件通过提供适合于表达测试的语言（以函数和方法的形式），并在测试失败时输出丰富的信息来帮助你构建和运行测试集合（测试套件，test suite）。 这些通常被称为测试运行器（test runner）。

一些代码比其他代码更容易测试。 通常，代码与外部交互的对象越多，建立用于测试它的上下文就越困难。 上一章中显示的编程风格，使用自包含的持久值而不是更改对象，通常很容易测试。

调试

当程序的运行结果不符合预期或在运行过程中产生错误时，你就会注意到程序出现问题了，下一步就是要推断问题出在什么地方。

有时错误很明显。错误消息会指出错误出现在程序的哪一行，只要稍加阅读错误描述及出错的那行代码，你一般就知道如何修正错误了。

但不总是这样。 有时触发问题的行，只是第一个地方，它以无效方式使用其他地方产生的奇怪的值。 如果你在前几章中已经解决了练习，你可能已经遇到过这种情况。

下面的示例代码尝试将一个整数转换成给定进制表示的字符串（十进制、二进制等），其原理是：不断循环取出最后一位数字，并将其除以基数（将最后一位数从数字中除去）。但该程序目前的输出表明程序中是存在bug的。

```
function numberToString(n, base = 10) {
  let result = "", sign = "";
  if (n < 0) {
    sign = "-";
    n = -n;
  }
  do {
    result = String(n % base) + result;
    n /= base;
  } while (n > 0);
  return sign + result;
}
console.log(numberToString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e-3181.3…
```

你可能已经发现程序运行结果不对了，不过先暂时装作不知道。我们知道程序运行出了问题，试图找出其原因。

这是一个地方，你必须抵制随机更改代码来查看它是否变得更好的冲动。 相反，要思考。 分析正在发生的事情，并提出为什么可能发生的理论。 然后，再做一些观察来检验这个理论 - 或者，如果你还没有理论，可以进一步观察来帮助你想出一个理论。

有目的地在程序中使用 `console.log` 来查看程序当前的运行状态，是一种不错的获取额外信息的方法。在本例中，我们希望 `n` 的值依次变为 13，1，然后是 0。让我们先在循环起始处输出 `n` 的值。

```
13
1.3
0.13
0.013
…
1.5e-323
```

没错。13 除以 10 并不会产生整数。我们不应该使用 `n/=base` ，而应该使用 `n=Math.floor(n/base)` ，使数字“右移”，这才是我们实际想要的结果。

使用 `console.log` 来查看程序行为的替代方法，是使用浏览器的调试器（debugger）功能。浏览器可以在代码的特定行上设置断点（breakpoint）。 当程序执行到带有断点的行时，它会暂停，并且你可以检查该点的绑定值。 我不会详细讨论，因为调试器在不同浏览器上有所不同，但请查看浏览器的开发人员工具或在 Web 上搜索来获取更多信息。

设置断点的另一种方法，是在程序中包含一个 `debugger` 语句（仅由该关键字组成）。 如果你的浏览器的开发人员工具是激活的，则只要程序达到这个语句，程序就会暂停。

错误传播

不幸的是，程序员不可能避免所有问题。 如果你的程序以任何方式与外部世界进行通信，则可能会导致输入格式错误，工作负荷过重或网络故障。

如果你只为自己编程，那么你就可以忽略这些问题直到它们发生。 但是如果你创建了一些将被其他人使用的东西，你通常希望程序比只是崩溃做得更好。 有时候，正确的做法是不择手段地继续运行。 在其他情况下，最好向用户报告出了什么问题然后放弃。 但无论在哪种情况下，该程序都必须积极采取措施来回应问题。

假设你有一个函数 `promptInteger`，要求用户输入一个整数并返回它。 如果用户输入 `"orange"`，它应该返回什么？

一种办法是返回一个特殊值，通常会使用 `null`，`undefined` 或 -1。

```
function promptNumber(question) {
  let result = Number(prompt(question, ""));
  if (Number.isNaN(result)) return null;
  else return result;
}

console.log(promptNumber("How many trees do you see?"));
```

现在，调用 `promptNumber` 的任何代码都必须检查是否实际读取了数字，否则必须以某种方式恢复 - 也许再次询问或填充默认值。 或者它可能会再次向它的调用者返回一个特殊值，表示它未能完成所要求的操作。

在很多情况下，当错误很常见并且调用者应该明确地考虑它们时，返回特殊值是表示错误的好方法。 但它确实有其不利之处。 首先，如果函数已经可能返回每一种可能的值呢？ 在这样的函数中，你必须做一些事情，比如将结果包装在一个对象中，以便能够区分成功与失败。

```
function lastElement(array) {
  if (array.length == 0) {
    return {failed: true};
  } else {
    return {element: array[array.length - 1]};
  }
}
```

返回特殊值的第二个问题是它可能产生非常笨拙的代码。 如果一段代码调用 `promptNumber` 10 次，则必须检查是否返回 `null` 10 次。 如果它对 `null` 的回应是简单地返回 `null` 本身，函数的调用者将不得不去检查它，以此类推。

异常

当函数无法正常工作时，我们只希望停止当前任务，并立即跳转到负责处理问题的位置。这就是异常处理的功能。

异常是一种当代码执行中遇到问题时，可以触发（或抛出）异常的机制，异常只是一个普通的值。触发异常类似于从函数中强制返回：异常不只跳出到当前函数中，还会跳出函数调用方，直到当前执行流初次调用函数的位置。这种方式被称为“堆栈展开（Unwinding the Stack）”。你可能还记得我们在第3章中介绍的函数调用栈，异常会减小堆栈的尺寸，并丢弃所有在缩减程序栈尺寸过程中遇到的函数调用上下文。

如果异常总是会将堆栈尺寸缩减到栈底，那么异常也就毫无用处了。它只不过是换了一种方式来彻底破坏你的程序罢了。异常真正强大的地方在于你可以在堆栈上设置一个“障碍物”，当异常缩减堆栈到达这个位置时会被捕获。一旦发现异常，你可以使用它来解决问题，然后继续运行该程序。

```
function promptDirection(question) {
  let result = prompt(question, "");
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new Error("Invalid direction: " + result);
}

function look() {
  if (promptDirection("Which way?") == "L") {
    return "a house";
  } else {
    return "two angry bears";
  }
}

try {
  console.log("You see", look());
} catch (error) {
  console.log("Something went wrong: " + error);
}
```

`throw` 关键字用于引发异常。 异常的捕获通过将一段代码包装在一个 `try` 块中，后跟关键字 `catch` 来完成。 当 `try` 块中的代码引发异常时，将求值 `catch` 块，并将括号中的名称绑定到异常值。 在 `catch` 块结束之后，或者 `try` 块结束并且没有问题时，程序在整个 `try / catch` 语句的下面继续执行。

在本例中，我们使用 `Error` 构造器来创建异常值。这是一个标准的 JavaScript 构造器，用于创建一个对象，包含 `message` 属性。在多数 JavaScript 环境中，构造器实例也会收集异常创建时的调用栈信息，即堆栈跟踪信息（Stack Trace）。该信息存储在 `stack` 属性中，对于调用问题有很大的帮助，我们可以从堆栈跟踪信息中得知问题发生的精确位置，即问题具体出现在哪个函数中，以及执行失败为止调用的其他函数链。

需要注意的是现在 `look` 函数可以完全忽略 `promptDirection` 出错的可能性。这就是使用异常的优势：只有在错误触发且必须处理的位置才需要错误处理代码。其间的函数可以忽略异常处理。

嗯，我们要讲解的理论知识差不多就这些了。

异常后清理

异常的效果是另一种控制流。 每个可能导致异常的操作（几乎每个函数调用和属性访问）都可能导致控制流突然离开你的代码。

这意味着当代码有几个副作用时，即使它的“常规”控制流看起来像它们总是会发生，但异常可能会阻止其中一些发生。

这是一些非常糟糕的银行代码。

```
const accounts = {
  a: 100,
  b: 0,
  c: 20
};

function getAccount() {
  let accountName = prompt("Enter an account name");
  if (!accounts.hasOwnProperty(accountName)) {
    throw new Error(`No such account: ${accountName}`);
  }
  return accountName;
}

function transfer(from, amount) {
  if (accounts[from] < amount) return;
  accounts[from] -= amount;
  accounts[getAccount()] += amount;
}
```

`transfer`函数将一笔钱从一个给定的账户转移到另一个账户，在此过程中询问另一个账户的名称。 如果给定一个无效的帐户名称，`getAccount`将引发异常。

但是`transfer`首先从帐户中删除资金，之后调用`getAccount`，之后将其添加到另一个帐户。 如果它在那个时候由异常中断，它就会让钱消失。

这段代码本来可以更智能一些，例如在开始转移资金之前调用`getAccount`。 但这样的问题往往以更微妙的方式出现。 即使是那些看起来不像是会抛出异常的函数，在特殊情况下，或者当他们包含程序员的错误时，也可能会这样。

解决这个问题的一个方法是使用更少的副作用。 同样，计算新值而不是改变现有数据的编程风格有所帮助。 如果一段代码在创建新值时停止运行，没有人会看到这个完成一半的值，并且没有问题。

但这并不总是实际的。 所以`try`语句具有另一个特性。 他们可能会跟着一个`finally`块，而不是`catch`块，也不是在它后面。 `finally`块会说“不管发生什么事，在尝试运行`try`块中的代码后，一定会运行这个代码。”

```
function transfer(from, amount) {
  if (accounts[from] < amount) return;
  let progress = 0;
  try {
    accounts[from] -= amount;
    progress = 1;
    accounts[getAccount()] += amount;
    progress = 2;
  } finally {
    if (progress == 1) {
      accounts[from] += amount;
    }
  }
}
```

这个版本的函数跟踪其进度，如果它在离开时注意到，它中止在创建不一致的程序状态的位置，则修复它造成的损害。

请注意，即使`finally`代码在异常退出`try`块时运行，它也不会影响异常。`finally`块运行后，堆栈继续展开。

即使异常出现在意外的地方，编写可靠运行的程序也非常困难。 很多人根本就不关心，而且由于异常通常针对异常情况而保留，因此问题可能很少发生，甚至从未被发现。 这是一件好事还是一件糟糕的事情，取决于软件执行失败时会造成多大的损害。

选择性捕获

当程序出现异常且异常未被捕获时，异常就会直接回退到栈顶，并由 JavaScript 环境来处理。其处理方式会根据环境的不同而不同。在浏览器中，错误描述通常会写入 JavaScript 控制台中（可以使用浏览器工具或开发者菜单来访问控制台）。我们将在第 20 章中讨论的，无浏览器的 JavaScript 环境 Node.js 对数据损坏更加谨慎。 当发生未处理的异常时，它会中止整个过程。

对于程序员的错误，让错误通行通常是最好的。 未处理的异常是表示糟糕的程序的合理方式，而在现代浏览器上，JavaScript 控制台为你提供了一些信息，有关在发生问题时堆栈上调用了哪些函数的。

对于在日常使用中发生的预期问题，因未处理的异常而崩溃是一种糟糕的策略。

语言的非法使用方式，比如引用一个不存在的绑定，在null中查询属性，或调用的对象不是函数最终都会引发异常。你可以像自己的异常一样捕获这些异常。

进入`catch`语句块时，我们只知道`try`体中引发了异常，但不知道引发了哪一类或哪一个异常。

JavaScript（很明显的疏漏）并未对选择性捕获异常提供良好的支持，要不捕获所有异常，要不什么都不捕获。这让你很容易假设，你得到的异常就是你在写`catch`时所考虑的异常。

但它也可能不是。 可能会违反其他假设，或者你可能引入了导致异常的 bug。 这是一个例子，它尝试持续调用 `promptDirection` ，直到它得到一个有效的答案：

```
for (;;) {
  try {
    let dir = promtDirection("Where?"); // ← typo!
    console.log("You chose ", dir);
    break;
  } catch (e) {
    console.log("Not a valid direction. Try again.");
  }
}
```

我们可以使用 `for (;;)` 循环体来创建一个无限循环，其自身永远不会停止运行。我们在用户给出有效的方向之后会跳出循环。但我们拼写错了 `promptDirection` ，因此会引发一个“未定义值”错误。由于 `catch` 块完全忽略了异常值，假定其知道问题所在，错将绑定错误信息当成错误输入。这样不仅会引发无限循环，而且会掩盖掉真正的错误消息——绑定名拼写错误。

一般而言，只有将抛出的异常重定位到其他地方进行处理时，我们才会捕获所有异常。比如说通过网络传输通知其他系统当前应用程序的崩溃信息。即便如此，我们也要注意编写的代码是否会将错误信息掩盖起来。

因此，我们转而会去捕获那些特殊类型的异常。我们可以在 `catch` 代码块中判断捕获到的异常是否就是我们期望处理的异常，如果不是则将其重新抛出。那么我们该如何辨别抛出异常的类型呢？

我们可以将它的 `message` 属性与我们所期望的错误信息进行比较。 但是，这是一种不稳定的编写代码的方式 - 我们将使用供人类使用的信息来做出程序化决策。 只要有人更改（或翻译）该消息，代码就会停止工作。

我们不如定义一个新的错误类型，并使用 `instanceof` 来识别异常。

```
class InputError extends Error {}

function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new InputError("Invalid direction: " + result);
}
```

新的错误类扩展了 `Error` 。 它没有定义它自己的构造器，这意味着它继承了 `Error` 构造器，它需要一个字符串消息作为参数。 事实上，它根本没有定义任何东西 - 这个类是空的。`InputError` 对象的行为与 `Error` 对象相似，只是它们的类不同，我们可以通过类来识别它们。

现在循环可以更仔细地捕捉它们。

```
for (;;) {
  try {
    let dir = promptDirection("Where?");
    console.log("You chose ", dir);
    break;
  } catch (e) {
    if (e instanceof InputError) {
      console.log("Not a valid direction. Try again.");
    } else {
      throw e;
    }
  }
}
```

这里的`catch`代码只会捕获`InputError`类型的异常，而其他类型的异常则不会在这里进行处理。如果又输入了不正确的值，那么系统会向用户准确报告错误——“绑定未定义”。

断言

断言（assertions）是程序内部的检查，用于验证某个东西是它应该是的方式。 它们并不是用于处理正常操作中可能出现的情况，而是发现程序员的错误。

例如，如果`firstElement`被描述为一个函数，永远不会在空数组上调用，我们可以这样写：

```
function firstElement(array) {
  if (array.length == 0) {
    throw new Error("firstElement called with []");
  }
  return array[0];
}
```

现在，它不会默默地返回未定义值（当你读取一个不存在的数组属性的时候），而是在你滥用它时立即干掉你的程序。 这使得这种错误不太可能被忽视，并且当它们发生时更容易找到它们的原因。

我不建议尝试为每种可能的不良输入编写断言。 这将是很多工作，并会产生非常杂乱的代码。 你会希望为很容易犯（或者你发现自己做过）的错误保留他们。

本章小结

错误和无效的输入十分常见。编程的一个重要部分是发现，诊断和修复错误。 如果你拥有自动化测试套件或向程序添加断言，则问题会变得更容易被注意。

我们常常需要使用优雅的方式来处理程序可控范围外的问题。如果问题可以就地解决，那么返回一个特殊的值来跟踪错误就是一个不错的解决方案。或者，异常也可能是可行的。

抛出异常会引发堆栈展开，直到遇到下一个封闭的 `try/catch` 块，或堆栈底部为止。`catch` 块捕获异常后，会将异常值赋予 `catch` 块，`catch` 块中应该验证异常是否是实际希望处理的异常，然后进行处理。为了有助于解决由于异常引起的不可预测的执行流，可以使用 `finally` 块来确保执行 `try` 块之后的代码。

习题

重试

假设有一个函数 `primitiveMultiply`，在 20% 的情况下将两个数相乘，在另外 80% 的情况下会触发 `MultiplicatorUnitFailure` 类型的异常。编写一个函数，调用这个容易出错的函数，不断尝试直到调用成功并返回结果为止。

确保只处理你期望的异常。

```
class MultiplicatorUnitFailure extends Error {}

function primitiveMultiply(a, b) {
  if (Math.random() < 0.2) {
    return a * b;
  } else {
    throw new MultiplicatorUnitFailure();
  }
}

function reliableMultiply(a, b) {
  // Your code here.
}

console.log(reliableMultiply(8, 8));
// → 64
```

上锁的箱子

考虑以下这个编写好的对象：

```
const box = {
  locked: true,
  unlock() { this.locked = false; },
  lock() { this.locked = true;  },
  _content: [],
  get content() {
    if (this.locked) throw new Error("Locked!");
    return this._content;
  }
};
```

这是一个带锁的箱子。其中有一个数组，但只有在箱子被解锁时，才可以访问数组。不允许直接访问 `_content` 属性。

编写一个名为 `withBoxUnlocked` 的函数，接受一个函数类型的参数，其作用是解锁箱子，执行该函数，无论是正常返回还是抛出异常，在 `withBoxUnlocked` 函数返回前都必须锁上箱子。

```
const box = {
  locked: true,
  unlock() { this.locked = false; },
  lock() { this.locked = true;  },
  _content: [],
  get content() {
    if (this.locked) throw new Error("Locked!");
    return this._content;
  }
};

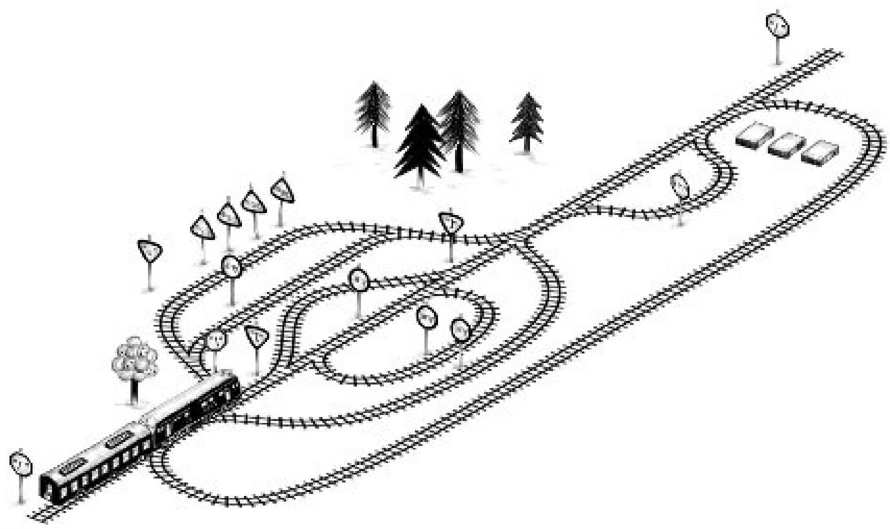
function withBoxUnlocked(body) {
  // Your code here.
}

withBoxUnlocked(function() {
  box.content.push("gold piece");
});

try {
  withBoxUnlocked(function() {
    throw new Error("Pirates on the horizon! Abort!");
  });
} catch (e) {
  console.log("Error raised:", e);
}
console.log(box.locked);
// → true
```

九、正则表达式

> 原文：Regular Expressions
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 一些人遇到问题时会认为，“我知道了，我会用正则表达式。”现在它们有两个问题了。
>
> Jamie Zawinski
>
> Yuan-Ma said, 'When you cut against the grain of the wood, much strength is needed. When you program against the grain of the problem, much code is needed.'
>
> Master Yuan-Ma，《The Book of Programming》



程序设计工具技术的发展与传播方式是在混乱中不断进化。在此过程中获胜的往往不是优雅或杰出的一方，而是那些瞄准主流市场，并能够填补市场需求的，或者碰巧与另一种成功的技术集成在一起的工具技术。

本章将会讨论正则表达式（regular expression）这种工具。正则表达式是一种描述字符串数据模式的方法。它们形成了一种小而独立的语言，也是 JavaScript 和许多其他语言和系统的一部分。

正则表达式虽然不易理解，但是功能非常强大。正则表达式的语法有点诡异，JavaScript 提供的程序设计接口也不太易用。但正则表达式的确是检查、处理字符串的强力工具。如果读者能够正确理解正则表达式，将会成为更高效的程序员。

创建正则表达式

正则表达式是一种对象类型。我们可以使用两种方法来构造正则表达式：一是使用 `RegExp` 构造器构造一个正则表达式对象；二是使用斜杠（`/`）字符将模式包围起来，生成一个字面值。

```
let re1 = new RegExp("abc");
let re2 = /abc/;
```

这两个正则表达式对象都表示相同的模式：字符 `a` 后紧跟一个 `b`，接着紧跟一个 `c`。

使用 `RegExp` 构造器时，需要将模式书写成普通的字符串，因此反斜杠的使用规则与往常相同。

第二种写法将模式写在斜杠之间，处理反斜杠的方式与第一种方法略有差别。首先，由于斜杠会结束整个模式，因此模式中包含斜杠时，需在斜杠前加上反斜杠。此外，如果反斜杠不是特殊字符代码（比如 `\n`）的一部分，则会保留反斜杠，不像字符串中会将其忽略，也不会改变模式的含义。一些字符，比如问号、加号在正则表达式中有特殊含义，如果你想要表示其字符本身，需要在字符前加上反斜杠。

```
let eighteenPlus = /eighteen\+/;
```

匹配测试

正则表达式对象有许多方法。其中最简单的就是 `test` 方法。`test` 方法接受用户传递的字符串，并返回一个布尔值，表示字符串中是否包含能与表达式模式匹配的字符串。

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

不包含特殊字符的正则表达式简单地表示一个字符序列。如果使用 `test` 测试字符串时，字符串中某处出现 `abc`（不一定在开头），则返回 `true`。

字符集

我们也可调用 `indexOf` 来找出字符串中是否包含 `abc`。正则表达式允许我们表达一些更复杂的模式。

假如我们想匹配任意数字。在正则表达式中，我们可以将一组字符放在两个方括号之间，该表达式可以匹配方括号中的任意字符。

下面两个表达式都可以匹配包含数字的字符串。

```
console.log(/[0123456789]/.test("in 1992"));
// → true
console.log(/[0-9]/.test("in 1992"));
// → true
```

我们可以在方括号中的两个字符间插入连字符（`-`），来指定一个字符范围，范围内的字符顺序由字符 Unicode 代码决定。在 Unicode 字符顺序中，0 到 9 是从左到右彼此相邻的（代码从48到57），因此 `[0-9]` 覆盖了这一范围内的所有字符，也就是说可以匹配任意数字。

许多常见字符组都有自己的内置简写。数字就是其中之一：`\ d` 与 `[0-9]` 表示相同的东西。

- `\d` 任意数字符号
- `\w` 字母和数字符号（单词符号）
- `\s` 任意空白符号（空格，制表符，换行符等类似符号）
- `\D` 非数字符号
- `\W` 非字母和数字符号
- `\S` 非空白符号
- `.` 除了换行符以外的任意符号

因此你可以使用下面的表达式匹配类似于 `30-01-2003 15:20` 这样的日期数字格式：

```
let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(dateTime.test("30-01-2003 15:20"));
// → true
console.log(dateTime.test("30-jan-2003 15:20"));
// → false
```

这个表达式看起来是不是非常糟糕？该表达式中一半都是反斜杠，影响读者的理解，使得读者难以揣摩表达式实际想要表达的模式。稍后我们会看到一个稍加改进的版本。

我们也可以将这些反斜杠代码用在方括号中。例如，`[\d.]` 匹配任意数字或一个句号。但是方括号中的句号会失去其特殊含义。其他特殊字符也是如此，比如 `+`。

你可以在左方括号后添加脱字符（`^`）来排除某个字符集，即表示不匹配这组字符中的任何字符。

```
let notBinary = /[^01]/;
console.log(notBinary.test("1100100010100110"));
// → false
console.log(notBinary.test("1100100010200110"));
// → true
```

部分模式重复

现在我们已经知道如何匹配一个数字。如果我们想匹配一个整数（一个或多个数字的序列），该如何处理呢？

在正则表达式某个元素后面添加一个加号（`+`），表示该元素至少重复一次。因此`/\d+/`可以匹配一个或多个数字字符。

```
console.log(/'\d+'/.test("'123'"));
// → true
console.log(/'\d+'/.test("''"));
// → false
console.log(/'\d*'/.test("'123'"));
// → true
console.log(/'\d*'/.test("''"));
// → true
```

星号（`*`）拥有类似含义，但是可以匹配模式不存在的情况。在正则表达式的元素后添加星号并不会导致正则表达式停止匹配该元素后面的字符。只有正则表达式无法找到可以匹配的文本时才会考虑匹配该元素从未出现的情况。

元素后面跟一个问号表示这部分模式“可选”，即模式可能出现 0 次或 1 次。下面的例子可以匹配`neighbour`（`u`出现1次），也可以匹配`neighbor`（`u`没有出现）。

```
let neighbor = /neighbou?r/;
console.log(neighbor.test("neighbour"));
// → true
console.log(neighbor.test("neighbor"));
// → true
```

我们可以使用花括号准确指明某个模式的出现次数。例如，在某个元素后加上`{4}`，则该模式需要出现且只能出现 4 次。也可以使用花括号指定一个范围：比如`{2,4}`表示该元素至少出现 2 次，至多出现 4 次。

这里给出另一个版本的正则表达式，可以匹配日期、月份、小时，每个数字都可以是一位或两位数字。这种形式更易于解释。

```
let dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(dateTime.test("30-1-2003 8:45"));
// → true
```

花括号中也可以省略逗号任意一侧的数字，表示不限制这一侧的数量。因此`{,5}`表示 0 到 5 次，而`{5,}`表示至少五次。

子表达式分组

为了一次性对多个元素使用`*`或者`+`，那么你必须使用圆括号，创建一个分组。对于后面的操作符来说，圆括号里的表达式算作单个元素。

```
let cartoonCrying = /boo+(hoo+)+/i;
console.log(cartoonCrying.test("Boohoooohoohooo"));
// → true
```

第一个和第二个`+`字符分别作用于`boo`与`hoo`的`o`字符，而第三个`+`字符则作用于整个元组（`hoo+`），可以匹配`hoo+`这种正则表达式出现一次及一次以上的情况。

示例中表达式末尾的i表示正则表达式不区分大小写，虽然模式中使用小写字母，但可以匹配输入字符串中的大写字母`B`。

匹配和分组

`test`方法是匹配正则表达式最简单的方法。该方法只负责判断字符串是否与某个模式匹配。正则表达式还有一个`exec`（执行，execute）方法，如果无法匹配模式则返回`null`，否则返回一个表示匹配字符串信息的对象。

```
let match = /\d+/.exec("one two 100");
console.log(match);
// → ["100"]
console.log(match.index);
// → 8
```

`exec`方法返回的对象包含index属性，表示字符串成功匹配的起始位置。除此之外，该对象看起来像（而且实际上就是）一个字符串数组，其首元素是与模式匹配的字符串——在上面的例子中就是我们查找的数字序列。

字符串也有一个类似的match方法。

```
console.log("one two 100".match(/\d+/));
// → ["100"]
```

若正则表达式包含使用圆括号包围的子表达式分组，与这些分组匹配的文本也会出现在数组中。第一个元素是与整个模式匹配的字符串，其后是与第一个分组匹配的部分字符串（表达式中第一次出现左圆括号的那部分），然后是第二个分组。

```
let quotedText = /'([^']*)'/;
console.log(quotedText.exec("she said 'hello'"));
// → ["'hello'", "hello"]
```

若分组最后没有匹配任何字符串（例如在元组后加上一个问号），结果数组中与该分组对应的元素将是`undefined`。类似的，若分组匹配了多个元素，则数组中只包含最后一个匹配项。

```
console.log(/bad(ly)?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

分组是提取部分字符串的实用特性。如果我们不只是想验证字符串中是否包含日期，还想将字符串中的日期字符串提取出来，并将其转换成等价的日期对象，那么我们可以使用圆括号包围那些匹配数字的模式字符串，并直接将日期从`exec`的结果中提取出来。

不过，我们暂且先讨论另一个话题——在 JavaScript 中存储日期和时间的内建方法。

日期类

JavaScript 提供了用于表示日期的标准类，我们甚至可以用其表示时间点。该类型名为`Date`。如果使用`new`创建一个`Date`对象，你会得到当前的日期和时间。

```
console.log(new Date());
// → Mon Nov 13 2017 16:19:11 GMT+0100 (CET)
```

你也可以创建表示特定时间的对象。

```
console.log(new Date(2009, 11, 9));
// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));
// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

JavaScript 中约定是：使用从 0 开始的数字表示月份（因此使用 11 表示 12 月），而使用从1开始的数字表示日期。这非常容易令人混淆。要注意这个细节。

构造器的后四个参数（小时、分钟、秒、毫秒）是可选的，如果用户没有指定这些参数，则参数的值默认为 0。

时间戳存储为 UTC 时区中 1970 年以来的毫秒数。 这遵循一个由“Unix 时间”设定的约定，该约定是在那个时候发明的。 你可以对 1970 年以前的时间使用负数。 日期对象上的`getTime`方法返回这个数字。 你可以想象它会很大。

```
console.log(new Date(2013, 11, 19).getTime());
// → 1387407600000
console.log(new Date(1387407600000));
// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)
```

如果你为`Date`构造器指定了一个参数，构造器会将该参数看成毫秒数。你可以创建一个新的`Date`对象，并调用`getTime`方法，或调用`Date.now()`函数来获取当前时间对应的毫秒数。

`Date`对象提供了一些方法来提取时间中的某些数值，比如`getFullYear`、`getMonth`、`getDate`、`getHours`、`getMinutes`、`getSeconds`。除了`getFullYear`之外该对象还有一个`getYear`方法，会返回使用两位数字表示的年份（比如93 或 14），但很少用到。

通过在希望捕获的那部分模式字符串两边加上圆括号，我们可以从字符串中创建对应的`Date`对象。

```
function getDate(string) {
  let [_, day, month, year] =
    /(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);
  return new Date(year, month - 1, day);
}
console.log(getDate("30-1-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)
```

`_`（下划线）绑定被忽略，并且只用于跳过由`exec`返回的数组中的，完整匹配元素。

单词和字符串边界

不幸的是，`getDate`会从字符串`"100-1-30000"`中提取出一个无意义的日期——`00-1-3000`。正则表达式可以从字符串中的任何位置开始匹配，在我们的例子中，它从第二个字符开始匹配，到倒数第二个字符为止。

如果我们想要强制匹配整个字符串，可以使用`^`标记和`$`标记。脱字符表示输入字符串起始位置，美元符号表示字符串结束位置。因此`/^\d+$/`可以匹配整个由一个或多个数字组成的字符串，`/^!/`匹配任何以感叹号开头的字符串，而`/x^/`不匹配任何字符串（字符串起始位置之前不可能有字符`x`）。

另一方面，如果我们想要确保日期字符串起始结束位置在单词边界上，可以使用`\b`标记。所谓单词边界，指的是起始和结束位置都是单词字符（也就是`\w`代表的字符集合），而起始位置的前一个字符以及结束位置的后一个字符不是单词字符。

```
console.log(/cat/.test("concatenate"));
// → true
console.log(/\bcat\b/.test("concatenate"));
// → false
```

这里需要注意，边界标记并不匹配实际的字符，只在强制正则表达式满足模式中的条件时才进行匹配。

选项模式

假如我们不仅想知道文本中是否包含数字，还想知道数字之后是否跟着一个单词（`pig`、`cow`或`chicken`）或其复数形式。

那么我们可以编写三个正则表达式并轮流测试，但还有一种更好的方式。管道符号（`|`）表示从其左侧的模式和右侧的模式任意选择一个进行匹配。因此代码如下所示。

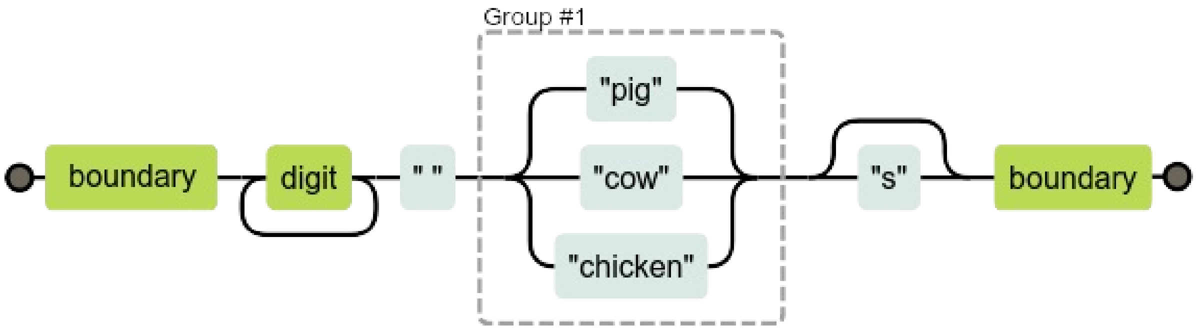
```
let animalCount = /\b\d+ (pig|cow|chicken)s?\b/;
console.log(animalCount.test("15 pigs"));
// → true
console.log(animalCount.test("15 pigchickens"));
// → false
```

小括号可用于限制管道符号选择的模式范围，而且你可以连续使用多个管道符号，表示从多于两个模式中选择一个备选项进行匹配。

匹配原理

从概念上讲，当你使用`exec`或`test`时，正则表达式引擎在你的字符串中寻找匹配，通过首先从字符串的开头匹配表达式，然后从第二个字符匹配表达式，直到它找到匹配或达到字符串的末尾。 它会返回找到的第一个匹配，或者根本找不到任何匹配。

为了进行实际的匹配，引擎会像处理流程图一样处理正则表达式。 这是上例中用于家畜表达式的图表：



如果我们可以找到一条从图表左侧通往图表右侧的路径，则可以说“表达式产生了匹配”。我们保存在字符串中的当前位置，每移动通过一个盒子，就验证当前位置之后的部分字符串是否与该盒子匹配。

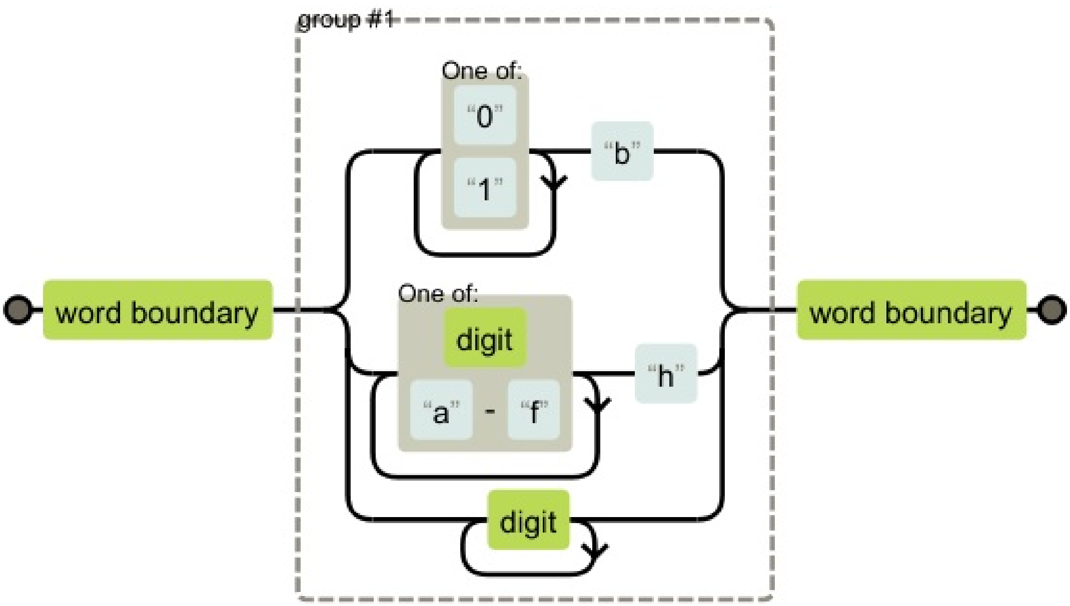
因此，如果我们尝试从位置 4 匹配`"the 3 pigs"`，大致会以如下的过程通过流程图：

- 在位置 4，有一个单词边界，因此我们通过第一个盒子。
- 依然在位置 4，我们找到一个数字，因此我们通过第二个盒子。
- 在位置 5，有一条路径循环回到第二个盒子（数字）之前，而另一条路径则移动到下一个盒子（单个空格字符）。由于这里是一个空格，而非数字，因此我们必须选择第二条路径。

- 我们目前在位置 6（`pig` 的起始位置），而表中有三路分支。这里看不到 `"cow"` 或 `"chicken"`，但我们看到了 `"pig"`，因此选择 `"pig"` 这条分支。
- 在位置 9（三路分支之后），有一条路径跳过了 `s` 这个盒子，直接到达最后的单词边界，另一条路径则匹配 `s`。这里有一个 `s` 字符，而非单词边界，因此我们通过 `s` 这个盒子。
- 我们在位置 10（字符串结尾），只能匹配单词边界。而字符串结尾可以看成一个单词边界，因此我们通过最后一个盒子，成功匹配字符串。

回溯

正则表达式 `/\b([01]+b|\d+|[\da-f]h)\b/` 可以匹配三种字符串：以 `b` 结尾的二进制数字，以 `h` 结尾的十六进制数字（即以 16 为进制，字母 `a` 到 `f` 表示数字 10 到 15），或者没有后缀字符的常规十进制数字。这是对应的图表。



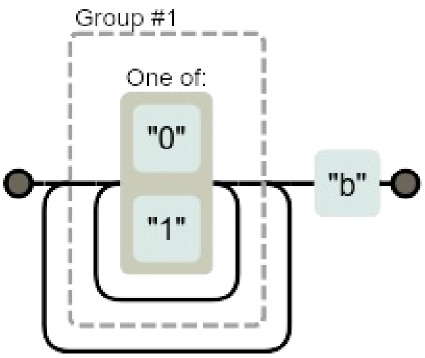
当匹配该表达式时，常常会发生一种情况：输入的字符串进入上方（二进制）分支的匹配过程，但输入中并不包含二进制数字。我们以匹配字符串 `"103"` 为例，匹配过程只有遇到字符 3 时才知道进入了错误分支。该字符串匹配我们给出的表达式，但没有匹配目前应当处于的分支。

因此匹配器执行“回溯”。进入一个分支时，匹配器会记住当前位置（在本例中，是在字符串起始，刚刚通过图中第一个表示边界的盒子），因此若当前分支无法匹配，可以回退并尝试另一条分支。对于字符串 `"103"`，遇到字符 3 之后，它会开始尝试匹配十六进制数字的分支，它会再次失败，因为数字后面没有 `h`。所以它尝试匹配进制数字的分支，由于这条分支可以匹配，因此匹配器最后的会返回十进制数的匹配信息。

一旦字符串与模式完全匹配，匹配器就会停止。这意味着多个分支都可能匹配一个字符串，但匹配器最后只会使用第一条分支（按照出现在正则表达式中的出现顺序排序）。

回溯也会发生在处理重复模式运算符（比如 `+` 和 `*`）时。如果使用 `"abcxe"` 匹配 `/^.*x/`，`.*` 部分，首先尝试匹配整个字符串，接着引擎发现匹配模式还需要一个字符 `x`。由于字符串结尾没有 `x`，因此 `*` 运算符尝试少匹配一个字符。但匹配器依然无法在 `abcx` 之后找到 `x` 字符，因此它会再次回溯，此时 `*` 运算符只匹配 `abc`。现在匹配器发现了所需的 `x`，接着报告从位置 0 到位置 4 匹配成功。

我们有可能编写需要大量回溯的正则表达式。当模式能够以许多种不同方式匹配输入的一部分时，这种问题就会出现。例如，若我们在编写匹配二进制数字的正则表达式时，一时糊涂，可能会写出诸如 `/([01]+)+b/` 之类的表达式。



若我们尝试匹配一些只由 0 与 1 组成的长序列，匹配器首先会不断执行内部循环，直到它发现没有数字为止。接下来匹配器注意到，这里不存在 `b`，因此向前回溯一个位置，开始执行外部循环，接着再次放弃，再次尝试执行一次内部循环。该过程会尝试这两个循环的所有可能路径。这意味着每多出一个字符，其工作量就会加倍。甚至只需较少的一堆字符，就可使匹配实际上永不停息地执行下去。

`replace` 方法

字符串有一个 `replace` 方法，该方法可用于将字符串中的一部分替换为另一个字符串。

```
console.log("papa".replace("p", "m"));
// → mapa
```

该方法第一个参数也可以是正则表达式，这种情况下会替换正则表达式首先匹配的部分字符串。若在正则表达式后追加 `g` 选项（全局，Global），该方法会替换字符串中所有匹配项，而不是只替换第一个。

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
// → Barabadar
```

如果 JavaScript 为 `replace` 添加一个额外参数，或提供另一个不同的方法（ `replaceAll` ），来区分替换一次匹配还是全部匹配，将会是较为明智的方案。遗憾的是，因为某些原因 JavaScript 依靠正则表达式的属性来区分替换行为。

如果我们在替换字符串中使用元组，就可以体现出 `replace` 方法的真实威力。例如，假设我们有一个规模很大的字符串，包含了人的名字，每个名字占据一行，名字格式为“姓，名”。若我们想要交换姓名，并移除中间的逗号（转变成“名，姓”这种格式），我们可以使用下面的代码：

```
console.log(
  "Liskov, Barbara\nMcCarthy, John\nWadler, Philip"
    .replace(/(\w+), (\w+)/g, "$2 $1"));
// → Barbara Liskov
//   John McCarthy
//   Philip Wadler
```

替换字符串中的 `$1` 和 `$2` 引用了模式中使用圆括号包裹的元组。 `$1` 会替换为第一个元组匹配的字符串， `$2` 会替换为第二个，依次类推，直到 `$9` 为止。也可以使用 `$&` 来引用整个匹配。

第二个参数不仅可以使用字符串，还可以使用一个函数。每次匹配时，都会调用函数并以匹配元组（也可以是匹配整体）作为参数，该函数返回值为需要插入的新字符串。

这里给出一个小示例：

```
let s = "the cia and fbi";
console.log(s.replace(/\b(fbi|cia)\b/g,
            str => str.toUpperCase()));
// → the CIA and FBI
```

这里给出另一个值得讨论的示例：

```
let stock = "1 lemon, 2 cabbages, and 101 eggs";
function minusOne(match, amount, unit) {
  amount = Number(amount) - 1;
  if (amount == 1) { // only one left, remove the 's'
    unit = unit.slice(0, unit.length - 1);
  } else if (amount == 0) {
    amount = "no";
  }
  return amount + " " + unit;
}
console.log(stock.replace(/(\d+) (\w+)/g, minusOne));
// → no lemon, 1 cabbage, and 100 eggs
```

该程序接受一个字符串，找出所有满足模式“一个数字紧跟着一个单词（数字和字母）”的字符串，返回时将捕获字符串中的数字减一。

元组 `(\d+)` 最后会变成函数中的 `amount` 参数，而 ·(\w+) `元组将会绑定 unit 。该函数将 amount 转换成数字（由于该参数是 `\d+`的匹配结果，因此此过程总是执行成功），并根据剩下 0 还是 1，决定如何做出调整。

贪婪模式

使用`replace`编写一个函数移除 JavaScript 代码中的所有注释也是可能的。这里我们尝试一下：

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*\*\//g, "");
}
console.log(stripComments("1 + /* 2 */3"));
// → 1 + 3
console.log(stripComments("x = 10;// ten!"));
// → x = 10;
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1  1
```

或运算符之前的部分匹配两个斜杠字符，后面跟着任意数量的非换行字符。多行注释部分较为复杂，我们使用`[^]`（任何非空字符集合）来匹配任意字符。我们这里无法使用句号，因为块注释可以跨行，句号无法匹配换行符。

但最后一行的输出显然有错。

为何？

在回溯一节中已经提到过，表达式中的`[^]*`部分会首先匹配所有它能匹配的部分。如果其行为引起模式的下一部分匹配失败，匹配器才会回溯一个字符，并再次尝试。在本例中，匹配器首先匹配整个剩余字符串，然后向前移动。匹配器回溯四个字符后，会找到*/，并完成匹配。这并非我们想要的结果。我们的意图是匹配单个注释，而非到达代码末尾并找到最后一个块注释的结束部分。

因为这种行为，所以我们说模式重复运算符（`+`、`*`、`?`和`{}`）是“贪婪”的，指的是这些运算符会尽量多地匹配它们可以匹配的字符，然后回溯。若读者在这些符号后加上一个问号（`+?`、`*?`、`??`、`{}?`），它们会变成非贪婪的，此时这些符号会尽量少地匹配字符，只有当剩下的模式无法匹配时才会多进行匹配。

而这便是我们想要的情况。通过让星号尽量少地匹配字符，我们可以匹配第一个`*/`，进而匹配一个块注释，而不会匹配过多内容。

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*?\*\//g, "");
}
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 + 1
```

对于使用了正则表达式的程序而言，其中出现的大量缺陷都可归咎于一个问题：在非贪婪模式效果更好时，无意间错用了贪婪运算符。若使用了模式重复运算符，请首先考虑一下是否可以使用非贪婪符号替代贪婪运算符。

动态创建`RegExp`对象

有些情况下，你无法在编写代码时准确知道需要匹配的模式。假设你想寻找文本片段中的用户名，并使用下划线字符将其包裹起来使其更显眼。由于你只有在程序运行时才知道姓名，因此你无法使用基于斜杠的记法。

但你可以构建一个字符串，并使用`RegExp`构造器根据该字符串构造正则表达式对象。

这里给出一个示例。

```
let name = "harry";
let text = "Harry is a suspicious character.";
let regexp = new RegExp("\\b(" + name + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → _Harry_ is a suspicious character.
```

由于我们创建正则表达式时使用的是普通字符串，而非使用斜杠包围的正则表达式，因此如果想创建`\b`边界，我们不得不使用两个反斜杠。`RegExp`构造器的第二个参数包含了正则表达式选项。在本例中，`"gi"`表示全局和不区分大小写。

但由于我们的用户是怪异的青少年，如果用户将名字设定为`"dea+hl[]rd"`，将会发生什么？这将会导致正则表达式变得没有意义，无法匹配用户名。

为了能够处理这种情况，我们可以在任何有特殊含义的字符前添加反斜杠。

```
let name = "dea+hl[]rd";
let text = "This dea+hl[]rd guy is super annoying.";
let escaped = name.replace(/[^\w\s]/g, "\\$&");
let regexp = new RegExp("\\b(" + escaped + ")\\b", "gi");
console.log(text.replace(regexp, "_$&_"));
// → This _dea+hl[]rd_ guy is super annoying.
```

`search`方法

字符串的`indexOf`方法不支持以正则表达式为参数。

但还有一个`search`方法，调用该方法时需要传递一个正则表达式。类似于`indexOf`，该方法会返回首先匹配的表达式的索引，若没有找到则返回–1。

```
console.log("  word".search(/\S/));
// → 2
console.log("    ".search(/\S/));
// → -1
```

遗憾的是，没有任何方式可以指定匹配的起始偏移（就像`indexOf`的第二个参数），而指定起始偏移这个功能是很实用的。

`lastIndex`属性

`exec` 方法同样没提供方便的方法来指定字符串中的起始匹配位置。但我们可以使用一种比较麻烦的方法来实现该功能。

正则表达式对象包含了一些属性。其中一个属性是 `source` ，该属性包含用于创建正则表达式的字符串。另一个属性是 `lastIndex` ，可以在极少数情况下控制下一次匹配的起始位置。

所谓的极少数情况，指的是当正则表达式启用了全局（ `g` ）或者粘性（ `y` ），并且使用 `exec` 匹配模式的时候。此外，另一个解决方案应该是向 `exec` 传递的额外参数，但 JavaScript 的正则表达式接口能设计得如此合理才是怪事。

```
let pattern = /y/g;
pattern.lastIndex = 3;
let match = pattern.exec("xyzzy");
console.log(match.index);
// → 4
console.log(pattern.lastIndex);
// → 5
```

如果成功匹配模式， `exec` 调用会自动更新 `lastIndex` 属性，来指向匹配字符串后的位置。如果无法匹配，会将 `lastIndex` 清零（就像新构建的正则表达式对象 `lastIndex` 属性为零一样）。

全局和粘性选项之间的区别在于，启用粘性时，仅当匹配直接从 `lastIndex` 开始时，搜索才会成功，而全局搜索中，它会搜索匹配可能起始的所有位置。

```
let global = /abc/g;
console.log(global.exec("xyz abc"));
// → ["abc"]
let sticky = /abc/y;
console.log(sticky.exec("xyz abc"));
// → null
```

对多个 `exec` 调用使用共享的正则表达式值时，这些 `lastIndex` 属性的自动更新可能会导致问题。 你的正则表达式可能意外地在之前的调用留下的索引处开始。

```
let digit = /\d/g;
console.log(digit.exec("here it is: 1"));
// → ["1"]
console.log(digit.exec("and now: 1"));
// → null
```

全局选项还有一个值得深思的效果，它会改变 `match` 匹配字符串的工作方式。如果调用 `match` 时使用了全局表达式，不像 `exec` 返回的数组， `match` 会找出所有匹配模式的字符串，并返回一个包含所有匹配字符串的数组。

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

因此使用全局正则表达式时需要倍加小心。只有以下几种情况中，你确实需要全局表达式即调用 `replace` 方法时，或是需要显示使用 `lastIndex` 时。这也基本是全局表达式唯一的应用场景了。

循环匹配

一个常见的事情是，找出字符串中所有模式的出现位置，这种情况下，我们可以在循环中使用 `lastIndex` 和 `exec` 访问匹配的对象。

```
let input = "A string with 3 numbers in it... 42 and 88.";
let number = /\b(\d+)\b/g;
let match;
while (match = number.exec(input)) {
  console.log("Found", match[0], "at", match.index);
}
// → Found 3 at 14
//   Found 42 at 33
//   Found 88 at 40
```

这里我们利用了赋值表达式的一个特性，该表达式的值就是被赋予的值。因此通过使用 `match=re.exec(input)` 作为 `while` 语句的条件，我们可以在每次迭代开始时执行匹配，将结果保存在变量中，当无法找到更多匹配的字符串时停止循环。

解析 `INI` 文件

为了总结一下本章介绍的内容，我们来看一下如何调用正则表达式来解决问题。假设我们编写一个程序从因特网上获取我们敌人的信息（这里我们实际上不会编写该程序，仅仅编写读取配置文件的那部分代码，对不起）。配置文件如下所示。

```
searchengine=https://duckduckgo.com/?q=$1
spitefulness=9.7

; comments are preceded by a semicolon...
; each section concerns an individual enemy
[larry]
fullname=Larry Doe
type=kindergarten bully
website=http://www.geocities.com/CapeCanaveral/11451

[davaeorn]
fullname=Davaeorn
type=evil wizard
outputdir=/home/marijn/enemies/davaeorn
```

该配置文件格式的语法规则如下所示（它是广泛使用的格式，我们通常称之为 `INI` 文件）：

- 忽略空行和以分号起始的行。
- 使用 `[]` 包围的行表示一个新的节（section）。

- 如果行中是一个标识符（包含字母和数字），后面跟着一个=字符，则表示向当前节添加选项。
- 其他的格式都是无效的。

我们的任务是将这样的字符串转换为一个对象，该对象的属性包含没有节的设置的字符串，和节的子对象的字符串，节的子对象也包含节的设置。

由于我们需要逐行处理这种格式的文件，因此预处理时最好将文件分割成一行行文本。我们使用第 6 章中的 `string.split("\n")` 来分割文件内容。但是一些操作系统并非使用换行符来分隔行，而是使用回车符加换行符（ `"\r\n"` ）。考虑到这点，我们也可以使用正则表达式作为 `split` 方法的参数，我们使用类似于 `/\r?\n/` 的正则表达式，这样可以同时支持 `"\n"` 和 `"\r\n"` 两种分隔符。

```
function parseINI(string) {
  // Start with an object to hold the top-level fields
  let currentSection = {name: null, fields: []};
  let categories = [currentSection];

  string.split(/\r?\n/).forEach(line => {
    let match;
    if (match = line.match(/^(\w+)=(.*)$/)) {
      section[match[1]] = match[2];
      section = result[match[1]] = {};
    } else if (!/^\s*(;.*)?$/.test(line)) {
      throw new Error("Line '" + line + "' is not valid.");
    }
  });

  return result;
}

console.log(parseINI(`
name=Vasilis
[address]
city=Tessaloniki`));
// → {name: "Vasilis", address: {city: "Tessaloniki"}}
```

代码遍历文件的行并构建一个对象。 顶部的属性直接存储在该对象中，而在节中找到的属性存储在单独的节对象中。 `section` 绑定指向当前节的对象。

有两种重要的行 - 节标题或属性行。 当一行是常规属性时，它将存储在当前节中。 当它是一个节标题时，创建一个新的节对象，并设置 `section` 来指向它。

这里需要注意，我们反复使用 `^` 和 `$` 确保表达式匹配整行，而非一行中的一部分。如果不使用这两个符号，大多数情况下程序也可以正常工作，但在处理特定输入时，程序就会出现不合理的行为，我们一般很难发现这个缺陷的问题所在。

`if (match = string.match(...))` 类似于使用赋值作为 `while` 的条件的技巧。你通常不确定你对 `match` 的调用是否成功，所以你只能在测试它的 `if` 语句中访问结果对象。 为了不打破 `else if` 形式的令人愉快的链条，我们将匹配结果赋给一个绑定，并立即使用该赋值作为 `if` 语句的测试。

国际化字符

由于 JavaScript 最初的实现非常简单，而且这种简单的处理方式后来也成了标准，因此 JavaScript 正则表达式处理非英语字符时非常无力。例如，就 JavaScript 的正则表达式而言，“单词字符”只是 26 个拉丁字母（大写和小写）和数字，而且由于某些原因还包括下划线字符。像 `α` 或 `β` 这种明显的单词字符，则无法匹配 `\w` （会匹配大写的 `\W` ，因为它们属于非单词字符）。

由于奇怪的历史性意外， `\s` （空白字符）则没有这种问题，会匹配所有 Unicode 标准中规定的空白字符，包括不间断空格和蒙古文元音分隔符。

另一个问题是，默认情况下，正则表达式使用代码单元，而不是实际的字符，正如第 5 章中所讨论的那样。 这意味着由两个代码单元组成的字符表现很奇怪。

```
console.log(/\ud83c\udf4e{3}/.test("\ud83c\udf4e\ud83c\udf4e\ud83c\udf4e"));
// → false
console.log(/<.>/.test("<\ud83c\udf39>"));
// → false
console.log(/<.>/u.test("<\ud83c\udf39>"));
// → true
```

问题是第一行中的 `"\ud83c\udf4e"` （emoji 苹果）被视为两个代码单元，而 `{3}` 部分仅适用于第二个。 与之类似，点匹配单个代码单元，而不是组成玫瑰 emoji 符号的两个代码单元。

你必须在正则表达式中添加一个 `u` 选项（表示 Unicode），才能正确处理这些字符。 不幸的是，错误的行为仍然是默认行为，因为改变它可能会导致依赖于它的现有代码出现问题。

尽管这是刚刚标准化的，在撰写本文时尚未得到广泛支持，但可以在正则表达式中使用 `\p` （必须启用 Unicode 选项）以匹配 Unicode 标准分配了给定属性的所有字符。

```
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
console.log(/\p{Alphabetic}/u.test("α"));
// → true
console.log(/\p{Alphabetic}/u.test("!"));
// → false
```

Unicode 定义了许多有用的属性，尽管找到你需要的属性可能并不总是没有意义。 你可以使用 `\p{Property=Value}` 符号来匹配任何具有该属性的给定值的字符。 如果属性名称保持不变，如 `\p{Name}` 中那样，名称被假定为二元属性，如 `Alphabetic` ，或者类别，如 `Number` 。

本章小结

正则表达式是表示字符串模式的对象，使用自己的语言来表达这些模式：

- `/abc/` ：字符序列

- `/[abc]/` ：字符集中的任何字符
- `/[^abc]/` ：不在字符集中的任何字符
- `/[0-9]/` ：字符范围内的任何字符
- `/x+/` ：出现一次或多次
- `/x+?/` ：出现一次或多次，非贪婪模式
- `/x*/` ：出现零次或多次
- `/x??/` ：出现零次或多次，非贪婪模式
- `/x{2，4}/` ：出现两次到四次
- `/(abc)/` ：元组
- `/a|b|c/` ：匹配任意一个模式
- `/\d/` ：数字字符
- `/\w/` ：字母和数字字符（单词字符）
- `/\s/` ：任意空白字符
- `/./` ：任意字符（除换行符外）
- `/\b/` ：单词边界
- `/^/` ：输入起始位置
- `/$/` ：输入结束位置

正则表达式有一个`test`方法来测试给定的字符串是否匹配它。 它还有一个`exec`方法，当找到匹配项时，返回一个包含所有匹配组的数组。 这样的数组有一个`index`属性，用于表明匹配开始的位置。

字符串有一个`match`方法来对正确表达式匹配它们，以及`search`方法来搜索字符串，只返回匹配的起始位置。 他们的`replace`方法可以用替换字符串或函数替换模式匹配。

正则表达式拥有选项，这些选项写在闭合斜线后面。 `i`选项使匹配不区分大小写。 `g`选项使表达式成为全聚德，除此之外，它使`replace`方法替换所有实例，而不是第一个。 `y`选项使它变为粘性，这意味着它在搜索匹配时不会向前搜索并跳过部分字符串。 `u`选项开启 Unicode 模式，该模式解决了处理占用两个代码单元的字符时的一些问题。

正则表达式是难以驾驭的强力工具。它可以简化一些任务，但用到一些复杂问题上时也会难以控制管理。想要学会使用正则表达式的重要一点是：不要将其用到无法干净地表达为正则表达式的问题。

习题

在做本章习题时，读者不可避免地会对一些正则表达式的莫名其妙的行为感到困惑，因而备受挫折。读者可以使用类似于 http://debuggex.com/ 这样的在线学习工具，将你想编写的正则表达式可视化，并试验其对不同输入字符串的响应。

RegexpGolf

Code Golf 是一种游戏，尝试尽量用最少的字符来描述特定程序。类似的，Regexp Golf 这种活动是编写尽量短小的正则表达式，来匹配给定模式（而且只能匹配给定模式）。

针对以下几项，编写正则表达式，测试给定的子串是否在字符串中出现。正则表达式匹配的字符串，应该只包含以下描述的子串之一。除非明显提到单词边界，否则千万不要担心边界问题。当你的表达式有效时，请检查一下能否让正则表达式更短小。

1. `car` 和 `cat`
2. `pop` 和 `prop`
3. `ferret` 、 `ferry` 和 `ferrari`
4. 以 `ious` 结尾的单词
5. 句号、冒号、分号之前的空白字符
6. 多于六个字母的单词
7. 不包含 `e` （或者 `E` ）的单词

需要帮助时，请参考本章总结中的表格。使用少量测试字符串来测试每个解决方案。

```
// Fill in the regular expressions

verify(/.../,
       ["my car", "bad cats"],
       ["camper", "high art"]);

verify(/.../,
       ["pop culture", "mad props"],
       ["plop", "prrrop"]]);

verify(/.../,
       ["ferret", "ferry", "ferrari"],
       ["ferrum", "transfer A"]);

verify(/.../,
       ["how delicious", "spacious room"],
       ["ruinous", "consciousness"]);

verify(/.../,
       ["bad punctuation ."],
       ["escape the period"]);

verify(/.../,
       ["hottentottententen"],
       ["no", "hotten totten tenten"]);

verify(/.../,
       ["red platypus", "wobbling nest"],
       ["earth bed", "learning ape", "BEET"]);


function verify(regexp, yes, no) {
  // Ignore unfinished exercises
  if (regexp.source == "...") return;
  for (let str of yes) if (!regexp.test(str)) {
    console.log(`Failure to match '${str}'`);
  }
  for (let str of no) if (regexp.test(str)) {
    console.log(`Unexpected match for '${str}'`);
  }
}
```

QuotingStyle

想象一下，你编写了一个故事，自始至终都使用单引号来标记对话。现在你想要将对话的引号替换成双引号，但不能替换在缩略形式中使用的单引号。

思考一下可以区分这两种引号用法的模式，并手动调用 `replace` 方法进行正确替换。

```
let text = "'I'm the cook,' he said, 'it's my job.'";
// Change this call.
console.log(text.replace(/A/g, "B"));
// → "I'm the cook," he said, "it's my job."
```

NumbersAgain

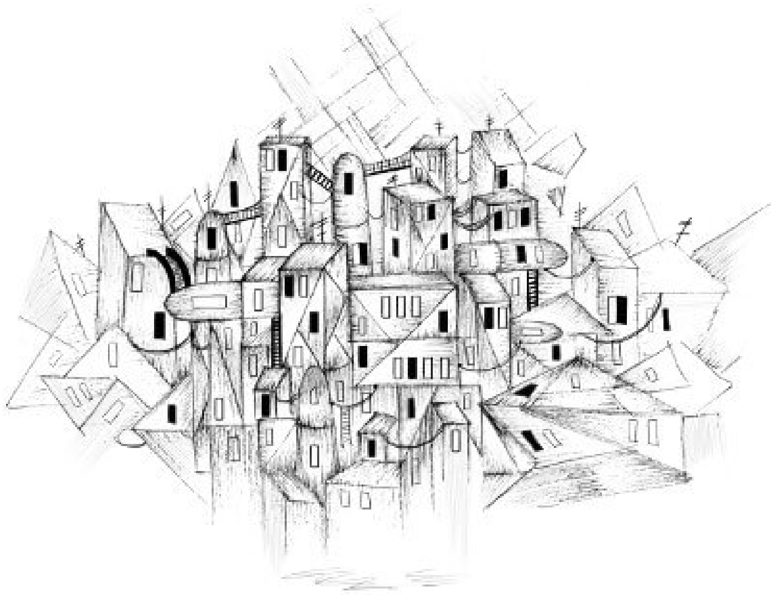
编写一个表达式，只匹配 JavaScript 风格的数字。支持数字前可选的正号与负号、十进制小数点、指数计数法（`5e-3` 或 `1E10`，指数前也需要支持可选的符号）。也请注意小数点前或小数点后的数字也是不必要的，但数字不能只有小数点。例如 `.5` 和 `5.` 都是合法的 JavaScript 数字，但单个点则不是。

```
// Fill in this regular expression.
let number = /^...$/;

// Tests:
for (let str of ["1", "-1", "+15", "1.55", ".5", "5.",
                 "1.3e2", "1E-4", "1e+12"]) {
  if (!number.test(str)) {
    console.log(`Failed to match '${str}'`);
  }
}
for (let str of ["1a", "+-1", "1.2.3", "1+1", "1e4.5",
                 ".5.", "1f5", "."]) {
  if (number.test(str)) {
    console.log(`Incorrectly accepted '${str}'`);
  }
}
```

十、模块

> 原文：Modules
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 编写易于删除，而不是易于扩展的代码。
>
> Tef，《Programming is Terrible》



理想的程序拥有清晰的结构。 它的工作方式很容易解释，每个部分都起到明确的作用。

典型的真实程序会有机地增长。 新功能随着新需求的出现而增加。 构建和维护结构是额外的工作，只有在下一次有人参与该计划时，才会得到回报。 所以它易于忽视，并让程序的各个部分变得深深地纠缠在一起。

这导致了两个实际问题。 首先，这样的系统难以理解。 如果一切都可以接触到一切其它东西，那么很难单独观察任何给定的片段。 你不得不全面理解整个东西。 其次，如果你想在另一个场景中，使用这种程序的任何功能，比起试图从它的上下文中将它分离出来，重写它可能要容易。

术语“大泥球”通常用于这种大型，无结构的程序。 一切都粘在一起，当你试图挑选出一段代码时，整个东西就会分崩离析，你的手会变脏。

模块

模块试图避免这些问题。 模块是一个程序片段，规定了它依赖的其他部分，以及它为其他模块提供的功能（它的接口）。

模块接口与对象接口有许多共同之处，我们在第 6 章中看到。它们向外部世界提供模块的一部分，并使其余部分保持私有。 通过限制模块彼此交互的方式，系统变得更像积木，其中的组件通过明确定义的连接器进行交互，而不像泥浆一样，一切都混在一起。

模块之间的关系称为依赖关系。 当一个模块需要另一个模块的片段时，就说它依赖于这个模块。 当模块中明确规定了这个事实时，它可以用于确定，需要哪些其他模块才能使用给定的模块，并自动加载依赖关系。

为了以这种方式分离模块，每个模块需要它自己的私有作用域。

将你的 JavaScript 代码放入不同的文件，不能满足这些要求。 这些文件仍然共享相同的全局命名空间。 他们可以有意或无意干扰彼此的绑定。 依赖性结构仍不清楚。 我们将在本章后面看到，我们可以做得更好。

合适的模块结构可能难以为程序设计。 在你还在探索这个问题的阶段，尝试不同的事情来看看什么是可行的，你可能不想过多担心它，因为这可能让你分心。 一旦你有一些感觉可靠的东西，现在是后退一步并组织它的好时机。

包

从单独的片段中构建一个程序，并实际上能够独立运行这些片段的一个优点是，你可能能够在不同的程序中应用相同的部分。

但如何实现呢？假设我想在另一个程序中使用第 9 章中的 `parseINI` 函数。 如果清楚该函数依赖什么（在这种情况下什么都没有），我可以将所有必要的代码复制到我的新项目中并使用它。 但是，如果我在代码中发现错误，我可能会在当时正在使用的任何程序中将其修复，并忘记在其他程序中修复它。

一旦你开始复制代码，你很快就会发现，自己在浪费时间和精力来到处复制并使他们保持最新。

这就是包的登场时机。包是可分发（复制和安装）的一大块代码。 它可能包含一个或多个模块，并且具有关于它依赖于哪些其他包的信息。 一个包通常还附带说明它做什么的文档，以便那些不编写它的人仍然可以使用它。

在包中发现问题或添加新功能时，会将包更新。 现在依赖它的程序（也可能是包）可以升级到新版本。

以这种方式工作需要基础设施。 我们需要一个地方来存储和查找包，以及一个便利方式来安装和升级它们。 在 JavaScript 世界中，这个基础结构由 NPM 提供。

NPM 是两个东西：可下载（和上传）包的在线服务，以及可帮助你安装和管理它们的程序（与 Node.js 捆绑在一起）。

在撰写本文时，NPM 上有超过 50 万个不同的包。 其中很大一部分是垃圾，我应该提一下，但几乎所有有用的公开包都可以在那里找到。 例如，一个 INI 文件解析器，类似于我们在第 9 章中构建的那个，可以在包名称 `ini` 下找到。

第 20 章将介绍如何使用 `npm` 命令行程序在局部安装这些包。

使优质的包可供下载是非常有价值的。 这意味着我们通常可以避免重新创建一百人之前写过的程序，并在按下几个键时得到一个可靠，充分测试的实现。

软件的复制很便宜，所以一旦有人编写它，分发给其他人是一个高效的过程。但首先把它写出来是工作量，回应在代码中发现问题的人，或者想要提出新功能的人，是更大的工作量。

默认情况下，你拥有你编写的代码的版权，其他人只有经过你的许可才能使用它。但是因为有些人不错，而且由于发布好的软件可以使你在程序员中出名，所以许多包都会在许可证下发布，明确允许其他人使用它。

NPM 上的大多数代码都以这种方式授权。某些许可证要求你还要在相同许可证下发布基于那个包构建的代码。其他要求不高，只是要求在分发代码时保留许可证。 JavaScript 社区主要使用后一种许可证。使用其他人的包时，请确保你留意了他们的许可证。

即兴的模块

2015 年之前，JavaScript 语言没有内置的模块系统。 然而，尽管人们已经用 JavaScript 构建了十多年的大型系统，他们需要模块。

所以他们在语言之上设计了自己的模块系统。 你可以使用 JavaScript 函数创建局部作用域，并使用对象来表示模块接口。

这是一个模块，用于日期名称和数字之间的转换（由 `Date` 的 `getDay` 方法返回）。 它的接口由 `weekDay.name` 和 `weekDay.number` 组成，它将局部绑定名称隐藏在立即调用的函数表达式的作用域内。

```
const weekDay = function() {
  const names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"];
  return {
    name(number) { return names[number]; },
    number(name) { return names.indexOf(name); }
  };
}();

console.log(weekDay.name(weekDay.number("Sunday")));
// → Sunday
```

这种风格的模块在一定程度上提供了隔离，但它不声明依赖关系。 相反，它只是将其接口放入全局范围，并希望它的依赖关系（如果有的话）也这样做。 很长时间以来，这是 Web 编程中使用的主要方法，但现在它几乎已经过时。

如果我们想让依赖关系成为代码的一部分，我们必须控制依赖关系的加载。 实现它需要能够将字符串执行为代码。 JavaScript 可以做到这一点。

将数据执行为代码

有几种方法可以将数据（代码的字符串）作为当前程序的一部分运行。

最明显的方法是特殊运算符 `eval`，它将在当前作用域内执行一个字符串。 这通常是一个坏主意，因为它破坏了作用域通常拥有的一些属性，比如易于预测给定名称所引用的绑定。

```
const x = 1;
function evalAndReturnX(code) {
  eval(code);
  return x;
}

console.log(evalAndReturnX("var x = 2"));
// → 2
console.log(x);
// → 1
```

将数据解释为代码的不太可怕的方法，是使用 `Function` 构造器。 它有两个参数：一个包含逗号分隔的参数名称列表的字符串，和一个包含函数体的字符串。 它将代码封装在一个函数值中，以便它获得自己的作用域，并且不会对其他作用域做出奇怪的事情。

```
let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

这正是我们需要的模块系统。 我们可以将模块的代码包装在一个函数中，并将该函数的作用域用作模块作用域。

CommonJS

用于连接 JavaScript 模块的最广泛的方法称为 CommonJS 模块。 Node.js 使用它，并且是 NPM 上大多数包使用的系统。

CommonJS 模块的主要概念是称为 `require` 的函数。 当你使用依赖项的模块名称调用这个函数时，它会确保该模块已加载并返回其接口。

由于加载器将模块代码封装在一个函数中，模块自动得到它们自己的局部作用域。 他们所要做的就是，调用 `require` 来访问它们的依赖关系，并将它们的接口放在绑定到 `exports` 的对象中。

此示例模块提供了日期格式化功能。 它使用 NPM的两个包， `ordinal` 用于将数字转换为字符串，如 `"1st"` 和 `"2nd"` ，以及 `date-names` 用于获取星期和月份的英文名称。 它导出函数 `formatDate` ，它接受一个 `Date` 对象和一个模板字符串。

模板字符串可包含指明格式的代码，如 `YYYY` 用于全年， `Do` 用于每月的序数日。 你可以给它一个像 `"MMMM Do YYYY"` 这样的字符串,来获得像 `"November 22nd 2017"` 这样的输出。

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");

exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};
```

`ordinal` 的接口是单个函数，而 `date-names` 导出包含多个东西的对象 - `days` 和 `months` 是名称数组。 为导入的接口创建绑定时，解构是非常方便的。

该模块将其接口函数添加到 `exports` ，以便依赖它的模块可以访问它。 我们可以像这样使用模块：

```
const {formatDate} = require("./format-date");

console.log(formatDate(new Date(2017, 9, 13),
                       "dddd the Do"));
// → Friday the 13th
```

我们可以用最简单的形式定义 `require` ，如下所示：

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);
    let module = {exports: {}};
    require.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module);
  }
  return require.cache[name].exports;
}
```

在这段代码中，`readFile` 是一个构造函数，它读取一个文件并将其内容作为字符串返回。标准的 JavaScript 没有提供这样的功能，但是不同的 JavaScript 环境（如浏览器和 Node.js）提供了自己的访问文件的方式。这个例子只是假设 `readFile` 存在。

为了避免多次加载相同的模块，`require` 需要保存（缓存）已经加载的模块。被调用时，它首先检查所请求的模块是否已加载，如果没有，则加载它。这涉及到读取模块的代码，将其包装在一个函数中，然后调用它。

我们之前看到的 `ordinal` 包的接口不是一个对象，而是一个函数。 CommonJS 模块的特点是，尽管模块系统会为你创建一个空的接口对象（绑定到 `exports` ），但你可以通过覆盖 `module.exports` 来替换它。许多模块都这么做，以便导出单个值而不是接口对象。

通过将 `require` ， `exports` 和 `module` 定义为生成的包装函数的参数（并在调用它时传递适当的值），加载器确保这些绑定在模块的作用域中可用。

提供给 `require` 的字符串翻译为实际的文件名或网址的方式，在不同系统有所不同。 当它以 `"./"` 或 `"../"` 开头时，它通常被解释为相对于当前模块的文件名。 所以 `"./format-date"` 就是在同一个目录中，名为 `format-date.js` 的文件。

当名称不是相对的时，Node.js 将按照该名称查找已安装的包。 在本章的示例代码中，我们将把这些名称解释为 NPM 包的引用。 我们将在第 20 章详细介绍如何安装和使用 NPM 模块。

现在，我们不用编写自己的 INI 文件解析器，而是使用 NPM 中的某个：

```
const {parse} = require("ini");

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

ECMAScript 模块

CommonJS 模块很好用，并且与 NPM 一起，使 JavaScript 社区开始大规模共享代码。

但他们仍然是个简单粗暴的黑魔法。 例如，表示法有点笨拙 - 添加到 `exports` 的内容在局部作用域中不可用。 而且因为 `require` 是一个正常的函数调用，接受任何类型的参数，而不仅仅是字符串字面值，所以在不运行代码就很难确定模块的依赖关系。

这就是 2015 年的 JavaScript 标准引入了自己的不同模块系统的原因。 它通常被称为 ES 模块，其中 ES 代表 ECMAScript。 依赖和接口的主要概念保持不变，但细节不同。 首先，表示法现在已整合到该语言中。 你不用调用函数来访问依赖关系，而是使用特殊的 `import` 关键字。

```
import ordinal from "ordinal";
import {days, months} from "date-names";

export function formatDate(date, format) { /* ... */ }
```

同样，`export` 关键字用于导出东西。 它可以出现在函数，类或绑定定义（`let`，`const` 或 `var`）的前面。

ES 模块的接口不是单个值，而是一组命名绑定。 前面的模块将 `formatDate` 绑定到一个函数。 从另一个模块导入时，导入绑定而不是值，这意味着导出模块可以随时更改绑定的值，导入它的模块将看到其新值。

当有一个名为 `default` 的绑定时，它将被视为模块的主要导出值。 如果你在示例中导入了一个类似于 `ordinal` 的模块，而没有绑定名称周围的大括号，则会获得其默认绑定。 除了默认绑定之外，这些模块仍然可以以不同名称导出其他绑定。

为了创建默认导出，可以在表达式，函数声明或类声明之前编写 `export default`。

```
export default ["Winter", "Spring", "Summer", "Autumn"];
```

可以使用单词 `as` 重命名导入的绑定。

```
import {days as dayNames} from "date-names";

console.log(dayNames.length);
// → 7
```

另一个重要的区别是，ES 模块的导入发生在模块的脚本开始运行之前。 这意味着 `import` 声明可能不会出现在函数或块中，并且依赖项的名称只能是带引号的字符串，而不是任意的表达式。

在撰写本文时，JavaScript 社区正在采用这种模块风格。 但这是一个缓慢的过程。 在规定格式之后，花了几年的时间，浏览器和 Node.js 才开始支持它。 虽然他们现在几乎都支持它，但这种支持仍然存在问题，这些模块如何通过 NPM 分发的讨论仍在进行中。

许多项目使用 ES 模块编写，然后在发布时自动转换为其他格式。 我们正处于并行使用两个不同模块系统的过渡时期，并且能够读写任何一种之中的代码都很有用。

构建和打包

事实上，从技术上来说，许多 JavaScript 项目都不是用 JavaScript 编写的。有一些扩展被广泛使用，例如第 8 章中提到的类型检查方言。很久以前，在语言的某个计划性扩展添加到实际运行 JavaScript 的平台之前，人们就开始使用它了。

为此，他们编译他们的代码，将其从他们选择的 JavaScript 方言翻译成普通的旧式 JavaScript，甚至是过去的 JavaScript 版本，以便旧版浏览器可以运行它。

在网页中包含由 200 个不同文件组成的模块化程序，会产生它自己的问题。如果通过网络获取单个文件需要 50 毫秒，则加载整个程序需要 10 秒，或者如果可以同时加载多个文件，则可能需要一半。这浪费了很多时间。因为抓取一个大文件往往比抓取很多小文件要快，所以 Web 程序员已经开始使用工具，将它们发布到 Web 之前，将他们（费力分割成模块）的程序回滚成单个大文件。这些工具被称为打包器。

我们可以再深入一点。 除了文件的数量之外，文件的大小也决定了它们可以通过网络传输的速度。 因此，JavaScript 社区发明了压缩器。 通过自动删除注释和空白，重命名绑定以及用占用更少空间的等效代码替换代码段，这些工具使 JavaScript 程序变得更小。

因此，你在 NPM 包中找到的代码，或运行在网页上的代码，经历了多个转换阶段 - 从现代 JavaScript 转换为历史 JavaScript，从 ES 模块格式转换为 CommonJS，打包并压缩。 我们不会在本书中详细介绍这些工具，因为它们往往很无聊，并且变化很快。 请注意，你运行的 JavaScript 代码通常不是编写的代码。

模块设计

使程序结构化是编程的一个微妙的方面。 任何有价值的功能都可以用各种方式建模。

良好的程序设计是主观的 - 涉及到权衡和品味问题。 了解结构良好的设计的价值的最好方法，是阅读或处理大量程序，并注意哪些是有效的，哪些不是。 不要认为一个痛苦的混乱就是“它本来的方式”。 通过多加思考，你可以改善几乎所有事物的结构。

模块设计的一个方面是易用性。 如果你正在设计一些旨在由多人使用，或者甚至是你自己的东西，在三个月之内，当你记不住你所做的细节时，如果你的接口简单且可预测，这会有所帮助。

这可能意味着遵循现有的惯例。 `ini` 包是一个很好的例子。 此模块模仿标准 JSON 对象，通过提供`parse`和`stringify`（用于编写 INI 文件）函数，就像 JSON 一样，在字符串和普通对象之间进行转换。 所以接口很小且很熟悉，在你使用过一次后，你可能会记得如何使用它。

即使没有能模仿的标准函数或广泛使用的包，你也可以通过使用简单的数据结构，并执行单一的重点事项，来保持模块的可预测性。 例如，NPM 上的许多 INI 文件解析模块，提供了直接从硬盘读取文件并解析它的功能。 这使得在浏览器中不可能使用这些模块，因为我们没有文件系统的直接访问权，并且增加了复杂性，通过组合模块与某些文件读取功能，可以更好地解决它。

这指向了模块设计的另一个有用的方面 - 一些代码可以轻易与其他代码组合。比起执行带有副作用的复杂操作的更大的模块，计算值的核心模块适用于范围更广的程序。坚持从磁盘读取文件的 INI 文件读取器， 在文件内容来自其他来源的场景中是无用的。

与之相关，有状态的对象有时甚至是有用的，但是如果某件事可以用一个函数完成，就用一个函数。 NPM 上的几个 INI 文件读取器提供了一种接口风格，需要你先创建一个对象，然后将该文件加载到对象中，最后使用特定方法来获取结果。这种类型的东西在面向对象的传统中很常见，而且很糟糕。你不能调用单个函数来完成，你必须执行仪式，在各种状态中移动对象。而且由于数据现在封装在一个特定的对象类型中，与它交互的所有代码都必须知道该类型，从而产生不必要的相互依赖关系。

通常，定义新的数据结构是不可避免的 - 只有少数非常基本的数据结构由语言标准提供，并且许多类型的数据一定比数组或映射更复杂。 但是当数组足够时，使用数组。

一个稍微复杂的数据结构的示例是第 7 章的图。JavaScript 中没有一种明显的表示图的方式。 在那一章中，我们使用了一个对象，其属性保存了字符串数组 - 可以从某个节点到达的其他节点。

NPM 上有几种不同的寻路包，但他们都没有使用这种图的格式。 它们通常允许图的边带有权重，它是与其相关的成本或距离，这在我们的表示中是不可能的。

例如，存在`dijkstrajs`包。 一种著名的寻路方法，与我们的`findRoute`函数非常相似，它被称为迪科斯特拉（Dijkstra）算法，以首先编写它的艾兹格尔·迪科斯特拉（Edsger Dijkstra）命名。 `js`后缀通常会添加到包名称中，以表明它们用 JavaScript 编写。 这个`dijkstrajs`包使用类似于我们的图的格式，但是它不使用数组，而是使用对象，它的属性值是数字 - 边的权重。

所以如果我们想要使用这个包，我们必须确保我们的图以它期望的格式存储。 所有边的权重都相同，因为我们的简化模型将每条道路视为具有相同的成本（一个回合）。

```
const {find_path} = require("dijkstrajs");

let graph = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
}

console.log(find_path(graph, "Post Office", "Cabin"));
// → ["Post Office", "Alice's House", "Cabin"]
```

这可能是组合的障碍 - 当各种包使用不同的数据结构来描述类似的事情时，将它们组合起来很困难。 因此，如果你想要设计可组合性，请查找其他人使用的数据结构，并在可能的情况下遵循他们的示例。

总结

通过将代码分离成具有清晰接口和依赖关系的块，模块是更大的程序结构。 接口是模块中可以从其他模块看到的部分，依赖关系是它使用的其他模块。

由于 JavaScript 历史上并没有提供模块系统，因此 CommonJS 系统建立在它之上。 然后在某个时候，它确实有了一个内置系统，它现在与 CommonJS 系统不兼容。

包是可以自行分发的一段代码。 NPM 是 JavaScript 包的仓库。 你可以从上面下载各种有用的（和无用的）包。

练习

模块化机器人

这些是第 7 章的项目所创建的约束：

```
roads
buildGraph
roadGraph
VillageState
runRobot
randomPick
randomRobot
mailRoute
routeRobot
findRoute
goalOrientedRobot
```

如果你要将该项目编写为模块化程序，你会创建哪些模块？ 哪个模块依赖于哪个模块，以及它们的接口是什么样的？

哪些片段可能在 NPM 上找到？ 你愿意使用 NPM 包还是自己编写？

`roads` 模块

根据第 7 章中的示例编写 CommonJS 模块，该模块包含道路数组，并将表示它们的图数据结构导出为 `roadGraph`。 它应该依赖于一个模块 `./graph`，它导出一个函数 `buildGraph`，用于构建图。 该函数接受包含两个元素的数组（道路的起点和终点）。

```
// Add dependencies and exports

const roads = [
  "Alice's House-Bob's House",   "Alice's House-Cabin",
  "Alice's House-Post Office",   "Bob's House-Town Hall",
  "Daria's House-Ernie's House", "Daria's House-Town Hall",
  "Ernie's House-Grete's House", "Grete's House-Farm",
  "Grete's House-Shop",          "Marketplace-Farm",
  "Marketplace-Post Office",     "Marketplace-Shop",
  "Marketplace-Town Hall",       "Shop-Town Hall"
];
```

循环依赖

循环依赖是一种情况，其中模块 A 依赖于 B，并且 B 也直接或间接依赖于 A。许多模块系统完全禁止这种情况，因为无论你选择何种顺序来加载此类模块，都无法确保每个模块的依赖关系在它运行之前加载。

CommonJS 模块允许有限形式的循环依赖。只要这些模块不会替换它们的默认`exports`对象，并且在完成加载之后才能访问对方的接口，循环依赖就没有问题。

本章前面给出的`require`函数支持这种类型的循环依赖。你能看到它如何处理循环吗？当一个循环中的某个模块替代其默认`exports`对象时，会出现什么问题？

十一、异步编程

> 原文：Asynchronous Programming
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 孰能浊以澄？静之徐清；
>
> 孰能安以久？动之徐生。
>
> 老子，《道德经》

计算机的核心部分称为处理器，它执行构成我们程序的各个步骤。 到目前为止，我们看到的程序都是让处理器忙碌，直到他们完成工作。 处理数字的循环之类的东西，几乎完全取决于处理器的速度。

但是许多程序与处理器之外的东西交互。 例如，他们可能通过计算机网络进行通信或从硬盘请求数据 - 这比从内存获取数据要慢很多。

当发生这种事情时，让处理器处于闲置状态是可耻的 - 在此期间可以做一些其他工作。 某种程度上，它由你的操作系统处理，它将在多个正在运行的程序之间切换处理器。 但是，我们希望单个程序在等待网络请求时能做一些事情，这并没有什么帮助。

异步

在同步编程模型中，一次只发生一件事。 当你调用执行长时间操作的函数时，它只会在操作完成时返回，并且可以返回结果。 这会在你执行操作的时候停止你的程序。

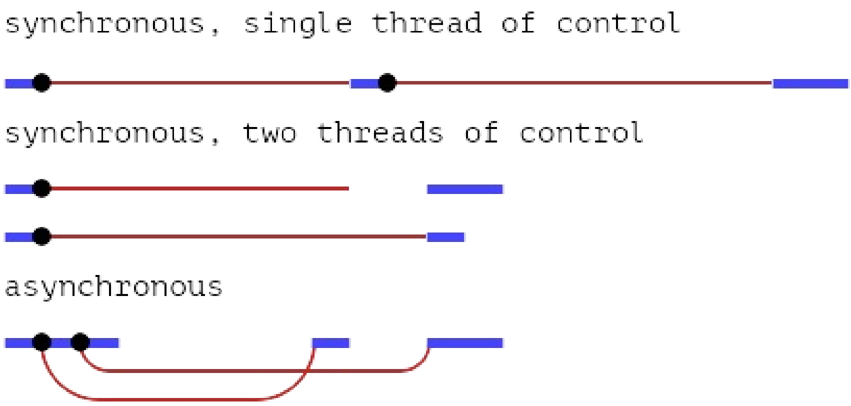
异步模型允许同时发生多个事件。 当你开始一个动作时，你的程序会继续运行。 当动作结束时，程序会受到通知并访问结果（例如从磁盘读取的数据）。

我们可以使用一个小例子来比较同步和异步编程：一个从网络获取两个资源然后合并结果的程序。

在同步环境中，只有在请求函数完成工作后，它才返回，执行此任务的最简单方法是逐个创建请求。 这有一个缺点，仅当第一个请求完成时，第二个请求才会启动。 所花费的总时间至少是两个响应时间的总和。

在同步系统中解决这个问题的方法是启动额外的控制线程。 线程是另一个正在运行的程序，它的执行可能会交叉在操作系统与其他程序当中 - 因为大多数现代计算机都包含多个处理器，所以多个线程甚至可能同时运行在不同的处理器上。 第二个线程可以启动第二个请求，然后两个线程等待它们的结果返回，之后它们重新同步来组合它们的结果。

在下图中，粗线表示程序正常花费运行的时间，细线表示等待网络所花费的时间。 在同步模型中，网络所花费的时间是给定控制线程的时间线的一部分。 在异步模型中，从概念上讲，启动网络操作会导致时间轴中出现分裂。 启动该动作的程序将继续运行，并且该动作将与其同时发生，并在程序结束时通知该程序。



另一种描述差异的方式是，等待动作完成在同步模型中是隐式的，而在异步模型中，在我们的控制之下，它是显式的。

异步性是个双刃剑。 它可以生成不适合直线控制模型的程序，但它也可以使直线控制的程序更加笨拙。 本章后面我们会看到一些方法来解决这种笨拙。

两种重要的 JavaScript 编程平台（浏览器和 Node.js）都可能需要一段时间的异步操作，而不是依赖线程。 由于使用线程进行编程非常困难（理解程序在同时执行多个事情时所做的事情要困难得多），这通常被认为是一件好事。

乌鸦科技

大多数人都知道乌鸦非常聪明。 他们可以使用工具，提前计划，记住事情，甚至可以互相沟通这些事情。

大多数人不知道的是，他们能够做一些事情，并且对我们隐藏得很好。我听说一个有声望的（但也有点古怪的）专家 corvids 认为，乌鸦技术并不落后于人类的技术，并且正在迎头赶上。

例如，许多乌鸦文明能够构建计算设备。 这些并不是电子的，就像人类的计算设备一样，但是它们操作微小昆虫的行动，这种昆虫是与白蚁密切相关的物种，它与乌鸦形成了共生关系。 鸟类为它们提供食物，对之对应，昆虫建立并操作复杂的殖民地，在其内部的生物的帮助下进行计算。

这些殖民地通常位于大而久远的鸟巢中。 鸟类和昆虫一起工作，建立一个球形粘土结构的网络，隐藏在巢的树枝之间，昆虫在其中生活和工作。

为了与其他设备通信，这些机器使用光信号。 鸟类在特殊的通讯茎中嵌入反光材料片段，昆虫校准这些反光材料将光线反射到另一个鸟巢，将数据编码为一系列快速闪光。 这意味着只有具有完整视觉连接的巢才能沟通。

我们的朋友 corvid 专家已经绘制了 Rhône 河畔的 Hières-sur-Amby 村的乌鸦鸟巢网络。 这张地图显示了鸟巢及其连接。

在一个令人震惊的趋同进化的例子中，乌鸦计算机运行 JavaScript。 在本章中，我们将为他们编写一些基本的网络函数。



回调

异步编程的一种方法是使执行慢动作的函数接受额外的参数，即回调函数。动作开始，当它结束时，使用结果调用回调函数。

例如，在 Node.js 和浏览器中都可用的`setTimeout`函数，等待给定的毫秒数（一秒为一千毫秒），然后调用一个函数。

```
setTimeout(() => console.log("Tick"), 500);
```

等待通常不是一种非常重要的工作，但在做一些事情时，例如更新动画或检查某件事是否花费比给定时间更长的时间，可能很有用。

使用回调在一行中执行多个异步操作，意味着你必须不断传递新函数来处理操作之后的计算延续。

大多数乌鸦鸟巢计算机都有一个长期的数据存储器，其中的信息刻在小树枝上，以便以后可以检索。雕刻或查找一段数据需要一些时间，所以长期存储的接口是异步的，并使用回调函数。

存储器按照名称存储 JSON 编码的数据片段。乌鸦可以存储它隐藏食物的地方的信息，其名称为 `"food caches"`，它可以包含指向其他数据片段的名称数组，描述实际的缓存。为了在 Big Oak 乌巢的存储器中查找食物缓存，乌鸦可以运行这样的代码：

```
import {bigOak} from "./crow-tech";

bigOak.readStorage("food caches", caches => {
  let firstCache = caches[0];
  bigOak.readStorage(firstCache, info => {
    console.log(info);
  });
});
```

（所有绑定名称和字符串都已从乌鸦语翻译成英语。）

这种编程风格是可行的，但缩进级别随着每个异步操作而增加，因为你最终会在另一个函数中。 做更复杂的事情，比如同时运行多个动作，会变得有点笨拙。

乌鸦鸟巢计算机为使用请求-响应对进行通信而构建。 这意味着一个鸟巢向另一个鸟巢发送消息，然后它立即返回一个消息，确认收到，并可能包括对消息中提出的问题的回复。

每条消息都标有一个类型，它决定了它的处理方式。 我们的代码可以为特定的请求类型定义处理器，并且当这样的请求到达时，调用处理器来产生响应。

`"./crow-tech"` 模块所导出的接口为通信提供基于回调的函数。 鸟巢拥有 `send` 方法来发送请求。 它接受目标鸟巢的名称，请求的类型和请求的内容作为它的前三个参数，以及一个用于调用的函数，作为其第四个和最后一个参数，当响应到达时调用。

```
bigOak.send("Cow Pasture", "note", "Let's caw loudly at 7PM",
            () => console.log("Note delivered."));
```

但为了使鸟巢能够接收该请求，我们首先必须定义名为 `"note"` 的请求类型。 处理请求的代码不仅要在这台鸟巢计算机上运行，而且还要运行在所有可以接收此类消息的鸟巢上。 我们只假定一只乌鸦飞过去，并将我们的处理器代码安装在所有的鸟巢中。

```
import {defineRequestType} from "./crow-tech";

defineRequestType("note", (nest, content, source, done) => {
  console.log(`${nest.name} received note: ${content}`);
  done();
});
```

`defineRequestType` 函数定义了一种新的请求类型。该示例添加了对 `"note"` 请求的支持，它只是向给定的鸟巢发送备注。我们的实现调用 `console.log`，以便我们可以验证请求到达。鸟巢有 `name` 属性，保存他们的名字。

给 `handler` 的第四个参数done，是一个回调函数，它在完成请求时必须调用。如果我们使用了处理器的返回值作为响应值，那么这意味着请求处理器本身不能执行异步操作。执行异步工作的函数通常会在完成工作之前返回，安排回调函数在完成时调用。所以我们需要一些异步机制 - 在这种情况下是另一个回调函数 - 在响应可用时发出信号。

某种程度上，异步性是传染的。任何调用异步的函数的函数，本身都必须是异步的，使用回调或类似的机制来传递其结果。调用回调函数比简单地返回一个值更容易出错，所以以这种方式构建程序的较大部分并不是很好。

Promise

当这些概念可以用值表示时，处理抽象概念通常更容易。 在异步操作的情况下，你不需要安排将来某个时候调用的函数，而是返回一个代表这个未来事件的对象。

这是标准类 `Promise` 的用途。 `Promise` 是一种异步行为，可以在某个时刻完成并产生一个值。 当值可用时，它能够通知任何感兴趣的人。

创建 `Promise` 的最简单方法是调用 `Promise.resolve`。 这个函数确保你给它的值包含在一个 `Promise` 中。 如果它已经是 `Promise`，那么仅仅返回它 - 否则，你会得到一个新的 `Promise`，并使用你的值立即结束。

```
let fifteen = Promise.resolve(15);
fifteen.then(value => console.log(`Got ${value}`));
// → Got 15
```

为了获得 `Promise` 的结果，可以使用它的 `then` 方法。 它注册了一个回调函数，当 `Promise` 解析并产生一个值时被调用。 你可以将多个回调添加到单个 `Promise` 中，即使在 `Promise` 解析（完成）后添加它们，它们也会被调用。

但那不是 `then` 方法所做的一切。 它返回另一个 `Promise`，它解析处理器函数返回的值，或者如果返回 `Promise`，则等待该 `Promise`，然后解析为结果。

将 `Promise` 视为一种手段，将值转化为异步现实，是有用处的。 一个正常的值就在那里。promised 的值是未来可能存在或可能出现的值。 根据 `Promise` 定义的计算对这些包装值起作用，并在值可用时异步执行。

为了创建 `Promise`，你可以将 `Promise` 用作构造器。 它有一个有点奇怪的接口 - 构造器接受一个函数作为参数，它会立即调用，并传递一个函数来解析这个 `Promise`。 它以这种方式工作，而不是使用 `resolve` 方法，这样只有创建 `Promise` 的代码才能解析它。

这就是为 `readStorage` 函数创建基于 `Promise` 的接口的方式。

```
function storage(nest, name) {
  return new Promise(resolve => {
    nest.readStorage(name, result => resolve(result));
  });
}

storage(bigOak, "enemies")
  .then(value => console.log("Got", value));
```

这个异步函数返回一个有意义的值。 这是 `Promise` 的主要优点 - 它们简化了异步函数的使用。 基于 `Promise` 的函数不需要传递回调，而是类似于常规函数：它们将输入作为参数并返回它们的输出。 唯一的区别是输出可能还不可用。

故障

> 译者注：这段如果有配套代码会更容易理解，但是没有，所以凑合看吧。

常规的 JavaScript 计算可能会因抛出异常而失败。 异步计算经常需要类似的东西。 网络请求可能会失败，或者作为异步计算的一部分的某些代码，可能会引发异常。

异步编程的回调风格中最紧迫的问题之一是，确保将故障正确地报告给回调函数，是非常困难的。

一个广泛使用的约定是，回调函数的第一个参数用于指示操作失败，第二个参数包含操作成功时生成的值。 这种回调函数必须始终检查它们是否收到异常，并确保它们引起的任何问题，包括它们调用的函数所抛出的异常，都会被捕获并提供给正确的函数。

`Promise` 使这更容易。可以解决它们（操作成功完成）或拒绝（故障）。只有在操作成功时，才会调用解析处理器（使用 `then` 注册），并且拒绝会自动传播给由 `then` 返回的新 `Promise`。当一个处理器抛出一个异常时，这会自动使 `then` 调用产生的 `Promise` 被拒绝。因此，如果异步操作链中的任何元素失败，则整个链的结果被标记为拒绝，并且不会调用失败位置之后的任何常规处理器。

就像 `Promise` 的解析提供了一个值，拒绝它也提供了一个值，通常称为拒绝的原因。当处理器中的异常导致拒绝时，异常值将用作原因。同样，当处理器返回被拒绝的 `Promise` 时，拒绝流入下一个 `Promise`。 `Promise.reject` 函数会创建一个新的，立即被拒绝的 `Promise`。

为了明确地处理这种拒绝，`Promise` 有一个 `catch` 方法，用于注册一个处理器，当 `Promise` 被拒绝时被调用，类似于处理器处理正常解析的方式。 这也非常类似于 `then`，因为它返回一个新的 `Promise`，如果它正常解析，它将解析原始 `Promise` 的值，否则返回 `catch` 处理器的结果。 如果 `catch` 处理器抛出一个错误，新的 `Promise` 也被拒绝。

作为简写，`then` 还接受拒绝处理器作为第二个参数，因此你可以在单个方法调用中，装配这两种的处理器。

传递给`Promise`构造器的函数接收第二个参数，并与解析函数一起使用，它可以用来拒绝新的`Promise`。

通过调用`then`和`catch`创建的`Promise`值的链条，可以看作异步值或失败沿着它移动的流水线。 由于这种链条通过注册处理器来创建，因此每个链条都有一个成功处理器或与其关联的拒绝处理器（或两者都有）。 不匹配结果类型（成功或失败）的处理器将被忽略。 但是那些匹配的对象被调用，并且它们的结果决定了下一次会出现什么样的值 -- 返回非`Promise`值时成功，当它抛出异常时拒绝，并且当它返回其中一个时是`Promise`的结果。

就像环境处理未捕获的异常一样，JavaScript 环境可以检测未处理`Promise`拒绝的时候，并将其报告为错误。

网络是困难的

偶尔，乌鸦的镜像系统没有足够的光线来传输信号，或者有些东西阻挡了信号的路径。 信号可能发送了，但从未收到。

事实上，这只会导致提供给`send`的回调永远不会被调用，这可能会导致程序停止，而不会注意到问题。 如果在没有得到回应的特定时间段内，请求会超时并报告故障，那就很好。

通常情况下，传输故障是随机事故，例如汽车的前灯会干扰光信号，只需重试请求就可以使其成功。 所以，当我们处理它时，让我们的请求函数在放弃之前自动重试发送请求几次。

而且，既然我们已经确定`Promise`是一件好事，我们也会让我们的请求函数返回一个`Promise`。 对于他们可以表达的内容，回调和`Promise`是等同的。 基于回调的函数可以打包，来公开基于`Promise`的接口，反之亦然。

即使请求及其响应已成功传递，响应也可能表明失败 - 例如，如果请求尝试使用未定义的请求类型或处理器，会引发错误。 为了支持这个，`send`和`defineRequestType`遵循前面提到的惯例，其中传递给回调的第一个参数是故障原因，如果有的话，第二个参数是实际结果。

这些可以由我们的包装翻译成`Promise`的解析和拒绝。

```
class Timeout extends Error {}

function request(nest, target, type, content) {
  return new Promise((resolve, reject) => {
    let done = false;
    function attempt(n) {
      nest.send(target, type, content, (failed, value) => {
        done = true;
        if (failed) reject(failed);
        else resolve(value);
      });
      setTimeout(() => {
        if (done) return;
        else if (n < 3) attempt(n + 1);
        else reject(new Timeout("Timed out"));
      }, 250);
    }
    attempt(1);
  });
}
```

因为`Promise`只能解析（或拒绝）一次，所以这个是有效的。第一次调用`resolve`或`reject`会决定`Promise`的结果，并且任何进一步的调用（例如请求结束后到达的超时，或在另一个请求结束后返回的请求）都将被忽略。

为了构建异步循环，对于重试，我们需要使用递归函数 - 常规循环不允许我们停止并等待异步操作。`attempt`函数尝试发送请求一次。它还设置了超时，如果 250 毫秒后没有响应返回，则开始下一次尝试，或者如果这是第四次尝试，则以`Timeout`实例为理由拒绝该`Promise`。

每四分之一秒重试一次，一秒钟后没有响应就放弃，这绝对是任意的。甚至有可能，如果请求确实过来了，但处理器花费了更长时间，请求将被多次传递。我们会编写我们的处理器，并记住这个问题 - 重复的消息应该是无害的。

总的来说，我们现在不会建立一个世界级的，强大的网络。但没关系 - 在计算方面，乌鸦没有很高的预期。

为了完全隔离我们自己的回调，我们将继续，并为`defineRequestType`定义一个包装器，它允许处理器返回一个`Promise`或明确的值，并且连接到我们的回调。

```
function requestType(name, handler) {
  defineRequestType(name, (nest, content, source,
                           callback) => {
    try {
      Promise.resolve(handler(nest, content, source))
        .then(response => callback(null, response),
              failure => callback(failure));
    } catch (exception) {
      callback(exception);
    }
  });
}
```

如果处理器返回的值还不是`Promise`，`Promise.resolve`用于将转换为`Promise`。

请注意，处理器的调用必须包装在`try`块中，以确保直接引发的任何异常都会被提供给回调函数。 这很好地说明了使用原始回调正确处理错误的难度 - 很容易忘记正确处理类似的异常，如果不这样做，故障将无法报告给正确的回调。 `Promise`使其大部分是自动的，因此不易出错。

`Promise`的集合

每台鸟巢计算机在其`neighbors`属性中，都保存了传输距离内的其他鸟巢的数组。 为了检查当前哪些可以访问，你可以编写一个函数，尝试向每个鸟巢发送一个`"ping"`请求（一个简单地请求响应的请求），并查看哪些返回了。

在处理同时运行的`Promise`集合时，`Promise.all`函数可能很有用。 它返回一个`Promise`，等待数组中的所有`Promise`解析，然后解析这些`Promise`产生的值的数组（与原始数组的顺序相同）。 如果任何`Promise`被拒绝，`Promise.all`的结果本身被拒绝。

```
requestType("ping", () => "pong");

function availableNeighbors(nest) {
  let requests = nest.neighbors.map(neighbor => {
    return request(nest, neighbor, "ping")
      .then(() => true, () => false);
  });
  return Promise.all(requests).then(result => {
    return nest.neighbors.filter((_, i) => result[i]);
  });
}
```

当一个邻居不可用时，我们不希望整个组合`Promise`失败，因为那时我们仍然不知道任何事情。 因此，在邻居集合上映射一个函数，将它们变成请求`Promise`，并附加处理器，这些处理器使成功的请求产生`true`，拒绝的产生`false`。

在组合`Promise`的处理器中，`filter`用于从`neighbors`数组中删除对应值为`false`的元素。 这利用了一个事实，`filter`将当前元素的数组索引作为其过滤函数的第二个参数（`map`，`some`和类似的高阶数组方法也一样）。

网络泛洪

鸟巢仅仅可以邻居通信的事实，极大地减少了这个网络的实用性。

为了将信息广播到整个网络，一种解决方案是设置一种自动转发给邻居的请求。 然后这些邻居转发给它们的邻居，直到整个网络收到这个消息。

```
import {everywhere} from "./crow-tech";

everywhere(nest => {
  nest.state.gossip = [];
});

function sendGossip(nest, message, exceptFor = null) {
  nest.state.gossip.push(message);
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "gossip", message);
  }
}

requestType("gossip", (nest, message, source) => {
  if (nest.state.gossip.includes(message)) return;
  console.log(`${nest.name} received gossip '${
               message}' from ${source}`);
  sendGossip(nest, message, source);
});
```

为了避免永远在网络上发送相同的消息，每个鸟巢都保留一组已经看到的闲话字符串。 为了定义这个数组，我们使用 `everywhere` 函数（它在每个鸟巢上运行代码）向鸟巢的状态对象添加一个属性，这是我们将保存鸟巢局部状态的地方。

当一个鸟巢收到一个重复的闲话消息，它会忽略它。每个人都盲目重新发送这些消息时，这很可能发生。 但是当它收到一条新消息时，它会兴奋地告诉它的所有邻居，除了发送消息的那个邻居。

这将导致一条新的闲话通过网络传播，如在水中的墨水一样。 即使一些连接目前不工作，如果有一条通往指定鸟巢的替代路线，闲话将通过那里到达它。

这种网络通信方式称为泛洪 - 它用一条信息充满网络，直到所有节点都拥有它。

我们可以调用 `sendGossip` 看看村子里的消息流。

```
sendGossip(bigOak, "Kids with airgun in the park");
```

消息路由

如果给定节点想要与其他单个节点通信，泛洪不是一种非常有效的方法。 特别是当网络很大时，这会导致大量无用的数据传输。

另一种方法是为消息设置节点到节点的传输方式，直到它们到达目的地。 这样做的困难在于，它需要网络布局的知识。 为了向远方的鸟巢发送请求，有必要知道哪个邻近的鸟巢更靠近其目的地。 以错误的方向发送它不会有太大好处。

由于每个鸟巢只知道它的直接邻居，因此它没有计算路线所需的信息。 我们必须以某种方式，将这些连接的信息传播给所有鸟巢。 当放弃或建造新的鸟巢时，最好是允许它随时间改变的方式。

我们可以再次使用泛洪，但不检查给定的消息是否已经收到，而是检查对于给定鸟巢来说，邻居的新集合，是否匹配我们拥有的当前集合。

```
requestType("connections", (nest, {name, neighbors},
                            source) => {
  let connections = nest.state.connections;
  if (JSON.stringify(connections.get(name)) ==
      JSON.stringify(neighbors)) return;
  connections.set(name, neighbors);
  broadcastConnections(nest, name, source);
});

function broadcastConnections(nest, name, exceptFor = null) {
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "connections", {
      name,
      neighbors: nest.state.connections.get(name)
    });
  }
}

everywhere(nest => {
  nest.state.connections = new Map;
  nest.state.connections.set(nest.name, nest.neighbors);
  broadcastConnections(nest, nest.name);
});
```

该比较使用`JSON.stringify`，因为对象或数组上的`==`只有在两者完全相同时才返回`true`，这不是我们这里所需的。 比较 JSON 字符串是比较其内容的一种简单而有效的方式。

节点立即开始广播它们的连接，它们应该立即为每个鸟巢提供当前网络图的映射，除非有一些鸟巢完全无法到达。

你可以用图做的事情，就是找到里面的路径，就像我们在第 7 章中看到的那样。如果我们有一条通往消息目的地的路线，我们知道将它发送到哪个方向。

这个`findRoute`函数非常类似于第 7 章中的`findRoute`，它搜索到达网络中给定节点的路线。 但不是返回整个路线，而是返回下一步。 下一个鸟巢将使用它的有关网络的当前信息，来决定将消息发送到哪里。

```
function findRoute(from, to, connections) {
  let work = [{at: from, via: null}];
  for (let i = 0; i < work.length; i++) {
    let {at, via} = work[i];
    for (let next of connections.get(at) || []) {
      if (next == to) return via;
      if (!work.some(w => w.at == next)) {
        work.push({at: next, via: via || next});
      }
    }
  }
  return null;
}
```

现在我们可以建立一个可以发送长途信息的函数。 如果该消息被发送给直接邻居，它将照常发送。 如果不是，则将其封装在一个对象中，并使用 `"route"` 请求类型，将其发送到更接近目标的邻居，这将导致该邻居重复相同的行为。

```
function routeRequest(nest, target, type, content) {
  if (nest.neighbors.includes(target)) {
    return request(nest, target, type, content);
  } else {
    let via = findRoute(nest.name, target,
                        nest.state.connections);
    if (!via) throw new Error(`No route to ${target}`);
    return request(nest, via, "route",
                   {target, type, content});
  }
}

requestType("route", (nest, {target, type, content}) => {
  return routeRequest(nest, target, type, content);
});
```

我们现在可以将消息发送到教堂塔楼的鸟巢中，它的距离有四跳。

```
routeRequest(bigOak, "Church Tower", "note",
             "Incoming jackdaws!");
```

我们已经在原始通信系统的基础上构建了几层功能，来使其便于使用。 这是一个（尽管是简化的）真实计算机网络工作原理的很好的模型。

计算机网络的一个显着特点是它们不可靠 - 建立在它们之上的抽象可以提供帮助，但是不能抽象出网络故障。所以网络编程通常关于预测和处理故障。

`async` 函数

为了存储重要信息，据了解乌鸦在鸟巢中复制它。 这样，当一只鹰摧毁一个鸟巢时，信息不会丢失。

为了检索它自己的存储器中没有的信息，鸟巢计算机可能会询问网络中其他随机鸟巢，直到找到一个鸟巢计算机。

```
requestType("storage", (nest, name) => storage(nest, name));

function findInStorage(nest, name) {
  return storage(nest, name).then(found => {
    if (found != null) return found;
    else return findInRemoteStorage(nest, name);
  });
}

function network(nest) {
  return Array.from(nest.state.connections.keys());
}

function findInRemoteStorage(nest, name) {
  let sources = network(nest).filter(n => n != nest.name);
  function next() {
    if (sources.length == 0) {
      return Promise.reject(new Error("Not found"));
    } else {
      let source = sources[Math.floor(Math.random() *
                                      sources.length)];
      sources = sources.filter(n => n != source);
      return routeRequest(nest, source, "storage", name)
        .then(value => value != null ? value : next(),
              next);
    }
  }
  return next();
}
```

因为`connections`是一个`Map`，`Object.keys`不起作用。它有一个`key`方法，但是它返回一个迭代器而不是数组。可以使用`Array.from`函数将迭代器（或可迭代对象）转换为数组。

即使使用`Promise`，这是一些相当笨拙的代码。多个异步操作以不清晰的方式链接在一起。我们再次需要一个递归函数（`next`）来建模鸟巢上的遍历。

代码实际上做的事情是完全线性的 - 在开始下一个动作之前，它总是等待先前的动作完成。在同步编程模型中，表达会更简单。

好消息是 JavaScript 允许你编写伪同步代码。异步函数是一种隐式返回`Promise`的函数，它可以在其主体中，以看起来同步的方式等待其他`Promise`。

我们可以像这样重写`findInStorage`：

```
async function findInStorage(nest, name) {
  let local = await storage(nest, name);
  if (local != null) return local;

  let sources = network(nest).filter(n => n != nest.name);
  while (sources.length > 0) {
    let source = sources[Math.floor(Math.random() *
                                    sources.length)];
    sources = sources.filter(n => n != source);
    try {
      let found = await routeRequest(nest, source, "storage",
                                     name);
      if (found != null) return found;
    } catch (_) {}
  }
  throw new Error("Not found");
}
```

异步函数由`function`关键字之前的`async`标记。方法也可以通过在名称前面编写`async`来做成异步的。当调用这样的函数或方法时，它返回一个`Promise`。只要主体返回了某些东西，这个`Promise`就解析了。如果它抛出异常，则`Promise`被拒绝。

```
findInStorage(bigOak, "events on 2017-12-21")
  .then(console.log);
```

在异步函数内部，`await`这个词可以放在表达式的前面，等待解`Promise`被解析，然后才能继续执行函数。

这样的函数不再像常规的 JavaScript 函数一样，从头到尾运行。相反，它可以在有任何带有`await`的地方冻结，并在稍后恢复。

对于有意义的异步代码，这种标记通常比直接使用`Promise`更方便。即使你需要做一些不适合同步模型的东西，比如同时执行多个动作，也很容易将`await`和直接使用`Promise`结合起来。

生成器

函数暂停然后再次恢复的能力，不是异步函数所独有的。JavaScript 也有一个称为生成器函数的特性。这些都是相似的，但没有`Promise`。

当用`function*`定义一个函数（在函数后面加星号）时，它就成为一个生成器。当你调用一个生成器时，它将返回一个迭代器，我们在第 6 章已经看到了它。

```
function* powers(n) {
  for (let current = n;; current *= n) {
    yield current;
  }
}

for (let power of powers(3)) {
  if (power > 50) break;
  console.log(power);
}
// → 3
// → 9
// → 27
```

最初，当你调用`powers`时，函数在开头被冻结。每次在迭代器上调用`next`时，函数都会运行，直到它碰到`yield`表达式，该表达式会暂停它，并使得产生的值成为由迭代器产生的下一个值。当函数返回时（示例中的那个永远不会），迭代器就结束了。

使用生成器函数时，编写迭代器通常要容易得多。可以用这个生成器编写`group`类的迭代器（来自第 6 章的练习）：

```
Group.prototype[Symbol.iterator] = function*() {
  for (let i = 0; i < this.members.length; i++) {
    yield this.members[i];
  }
};
```

不再需要创建一个对象来保存迭代状态 - 生成器每次 `yield` 时都会自动保存其本地状态。

这样的 `yield` 表达式可能仅仅直接出现在生成器函数本身中，而不是在你定义的内部函数中。 生成器在返回（ `yield` ）时保存的状态，只是它的本地环境和它 `yield` 的位置。

异步函数是一种特殊的生成器。 它在调用时会产生一个 `Promise` ，当它返回（完成）时被解析，并在抛出异常时被拒绝。 每当它 `yield` （ `await` ）一个 `Promise` 时，该 `Promise` 的结果（值或抛出的异常）就是 `await` 表达式的结果。

事件循环

异步程序是逐片段执行的。 每个片段可能会启动一些操作，并调度代码在操作完成或失败时执行。 在这些片段之间，该程序处于空闲状态，等待下一个动作。

所以回调函数不会直接被调度它们的代码调用。 如果我从一个函数中调用 `setTimeout` ，那么在调用回调函数时该函数已经返回。 当回调返回时，控制权不会回到调度它的函数。

异步行为发生在它自己的空函数调用堆栈上。 这是没有 `Promise` 的情况下，在异步代码之间管理异常很难的原因之一。 由于每个回调函数都是以几乎为空的堆栈开始，因此当它们抛出一个异常时，你的 `catch` 处理程序不会在堆栈中。

```
try {
  setTimeout(() => {
    throw new Error("Woosh");
  }, 20);
} catch (_) {
  // This will not run
  console.log("Caught!");
}
```

无论事件发生多么紧密（例如超时或传入请求），JavaScript 环境一次只能运行一个程序。 你可以把它看作在程序周围运行一个大循环，称为事件循环。 当没有什么可以做的时候，那个循环就会停止。 但随着事件来临，它们被添加到队列中，并且它们的代码被逐个执行。 由于没有两件事同时运行，运行缓慢的代码可能会延迟其他事件的处理。

这个例子设置了一个超时，但是之后占用时间，直到超时的预定时间点，导致超时延迟。

```
let start = Date.now();
setTimeout(() => {
  console.log("Timeout ran at", Date.now() - start);
}, 20);
while (Date.now() < start + 50) {}
console.log("Wasted time until", Date.now() - start);
// → Wasted time until 50
// → Timeout ran at 55
```

`Promise` 总是作为新事件来解析或拒绝。 即使已经解析了 `Promise` ，等待它会导致你的回调在当前脚本完成后运行，而不是立即执行。

```
Promise.resolve("Done").then(console.log);
console.log("Me first!");
// → Me first!
// → Done
```

在后面的章节中，我们将看到在事件循环中运行的，各种其他类型的事件。

异步的 bug

当你的程序同步运行时，除了那些程序本身所做的外，没有发生任何状态变化。 对于异步程序，这是不同的 - 它们在执行期间可能会有空白，这个时候其他代码可以运行。

我们来看一个例子。 我们乌鸦的爱好之一是计算整个村庄每年孵化的雏鸡数量。 鸟巢将这一数量存储在他们的存储器中。 下面的代码尝试枚举给定年份的所有鸟巢的计数。

```
function anyStorage(nest, source, name) {
  if (source == nest.name) return storage(nest, name);
  else return routeRequest(nest, source, "storage", name);
}

async function chicks(nest, year) {
  let list = "";
  await Promise.all(network(nest).map(async name => {
    list += `${name}: ${
      await anyStorage(nest, name, `chicks in ${year}`)
    }\n`;
  }));
  return list;
}
```

`async name =>` 部分展示了，通过将单词 `async` 放在它们前面，也可以使箭头函数变成异步的。

代码不会立即看上去有问题......它将异步箭头函数映射到鸟巢集合上，创建一组 `Promise` ，然后使用 `Promise.all` ，在返回它们构建的列表之前等待所有 `Promise` 。

但它有严重问题。 它总是只返回一行输出，列出响应最慢的鸟巢。

```
chicks(bigOak, 2017).then(console.log);
```

你能解释为什么吗？

问题在于 `+=` 操作符，它在语句开始执行时接受 `list` 的当前值，然后当 `await` 结束时，将 `list` 绑定设为该值加上新增的字符串。

但是在语句开始执行的时间和它完成的时间之间存在一个异步间隔。 `map` 表达式在任何内容添加到列表之前运行，因此每个 `+ =` 操作符都以一个空字符串开始，并在存储检索完成时结束，将 `list` 设置为单行列表 - 向空字符串添加那行的结果。

通过从映射的 `Promise` 中返回行，并对 `Promise.all` 的结果调用 `join` ，可以轻松避免这种情况，而不是通过更改绑定来构建列表。 像往常一样，计算新值比改变现有值的错误更少。

```
async function chicks(nest, year) {
  let lines = network(nest).map(async name => {
    return name + ": " +
      await anyStorage(nest, name, `chicks in ${year}`);
  });
  return (await Promise.all(lines)).join("\n");
}
```

像这样的错误很容易做出来，特别是在使用 `await` 时，你应该知道代码中的间隔在哪里出现。 JavaScript 的显式异步性（无论是通过回调， `Promise` 还是 `await` ）的一个优点是，发现这些间隔相对容易。

总结

异步编程可以表示等待长时间运行的动作，而不需要在这些动作期间冻结程序。 JavaScript 环境通常使用回调函数来实现这种编程风格，这些函数在动作完成时被调用。 事件循环调度这样的回调，使其在适当的时候依次被调用，以便它们的执行不会重叠。

`Promise` 和异步函数使异步编程更容易。 `Promise` 是一个对象，代表将来可能完成的操作。并且，异步函数使你可以像编写同步程序一样编写异步程序。

练习

跟踪手术刀

村里的乌鸦拥有一把老式的手术刀，他们偶尔会用于特殊的任务 - 比如说，切开纱门或包装。为了能够快速追踪到手术刀，每次将手术刀移动到另一个鸟巢时，将一个条目添加到拥有它和拿走它的鸟巢的存储器中，名称为 `"scalpel"` ，值为新的位置。

这意味着找到手术刀就是跟踪存储器条目的痕迹，直到你发现一个鸟巢指向它本身。

编写一个异步函数`locateScalpel`，它从它运行的鸟巢开始。你可以使用之前定义的`anyStorage`函数，来访问任意鸟巢中的存储器。手术刀已经移动了很长时间，你可能会认为每个鸟巢的数据存储器中都有一个`"scalpel"`条目。

接下来，再次写入相同的函数，而不使用`async`和`await`。

在两个版本中，请求故障是否正确显示为拒绝？如何实现？

```
async function locateScalpel(nest) {
  // Your code here.
}

function locateScalpel2(nest) {
  // Your code here.
}

locateScalpel(bigOak).then(console.log);
// → Butcher Shop
```

构建`Promise.all`

给定`Promise`的数组，`Promise.all`返回一个`Promise`，等待数组中的所有`Promise`完成。然后它成功，产生结果值的数组。如果数组中的一个`Promise`失败，这个`Promise`也失败，故障原因来自那个失败的`Promise`。

自己实现一个名为`Promise_all`的常规函数。

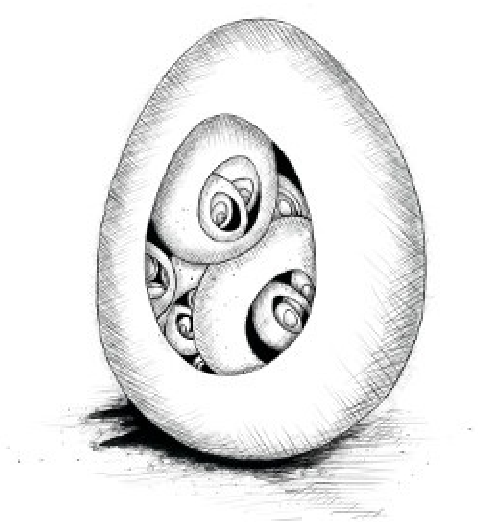
请记住，在`Promise`成功或失败后，它不能再次成功或失败，并且解析它的函数的进一步调用将被忽略。这可以简化你处理`Promise`的故障的方式。

```
function Promise_all(promises) {
  return new Promise((resolve, reject) => {
    // Your code here.
  });
}

// Test code.
Promise_all([]).then(array => {
  console.log("This should be []:", array);
});
function soon(val) {
  return new Promise(resolve => {
    setTimeout(() => resolve(val), Math.random() * 500);
  });
}
Promise_all([soon(1), soon(2), soon(3)]).then(array => {
  console.log("This should be [1, 2, 3]:", array);
});
Promise_all([soon(1), Promise.reject("X"), soon(3)])
  .then(array => {
    console.log("We should not get here");
  })
  .catch(error => {
    if (error != "X") {
      console.log("Unexpected failure:", error);
    }
  });
```

十二、项目：编程语言

> 原文：Project: A Programming Language
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 确定编程语言中的表达式含义的求值器只是另一个程序。
>
> Hal Abelson 和 Gerald Sussman，《计算机程序的构造和解释》



构建你自己的编程语言不仅简单（只要你的要求不要太高就好），而且对人富有启发。

希望通过本章的介绍，你能发现构建自己的编程语言其实并不是什么难事。我经常感到某些人的想法聪明无比，而且十分复杂，以至于我都不能完全理解。不过经过一段时间的阅读和实验，我就发现它们其实也并没有想象中那么复杂。

我们将创造一门名为 Egg 的编程语言。这是一门小巧而简单的语言，但是足够强大到能描述你所能想到的任何计算。它允许基于函数的简单抽象。

解析

程序设计语言中最直观的部分就是语法（syntax）或符号。解析器是一种程序，负责读入文本片段（包含程序的文本），并产生一系列与程序结构对应的数据结构。若文本不是一个合法程序，解析器应该指出错误。

我们的语言语法简单，而且具有一致性。Egg 中一切都是表达式。表达式可以是绑定名称、数字，或应用（application）。不仅函数调用属于应用，而且 `if` 和 `while` 之类的语言构造也属于应用。

为了确保解析器的简单性，Egg 中的字符串不支持反斜杠转义符之类的元素。字符串只是简单的字符序列（不包括双引号），并使用双引号包围起来。数值是数字序列。绑定名由任何非空白字符组成，并且在语法中不具有特殊含义。

应用的书写方式与 JavaScript 中一样，也是在一个表达式后添加一对括号，括号中可以包含任意数量的参数，参数之间使用逗号分隔。

```
do(define(x, 10),
   if(>(x, 5),
      print("large"),
      print("small")))
```

Egg 语言的一致性体现在：JavaScript 中的所有运算符（比如 `>`）在 Egg 中都是绑定，但是可以像其他函数一样调用。由于语法中没有语句块的概念，因此我们需要使用 `do` 结构来表示多个表达式的序列。

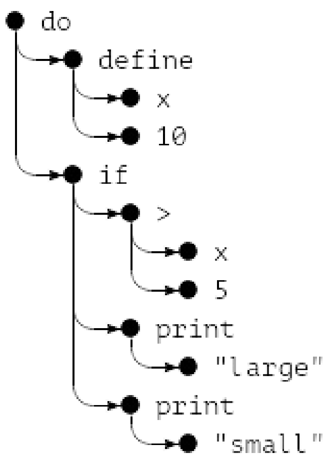
解析器的数据结构用于描述由表达式对象组成的程序，每个对象都包含一个表示表达式类型的 `type` 属性，除此以外还有其他描述对象内容的属性。

类型为 `"value"` 的表达式表示字符串和数字。它们的 `value` 属性包含对应的字符串和数字值。类型为 `"word"` 的表达式用于标识符（名称）。这类对象以字符串形式将标识符名称保存在 `name` 属性中。最后，类型为 `"apply"` 的表达式表示应用。该类型的对象有一个 `operator` 属性，指向其操作的表达式，还有一个 `args` 属性，持有参数表达式的数组。

上面代码中 `> (x, 5)` 这部分可以表达成如下形式：

```
{
  type: "apply",
  operator: {type: "word", name: ">"},
  args: [
    {type: "word", name: "x"},
    {type: "value", value: 5}
  ]
}
```

我们将这样一个数据结构称为表达式树。如果你将对象想象成点，将对象之间的连接想象成点之间的线，这个数据结构将会变成树形。表达式中还会包含其他表达式，被包含的表达式接着又会包含更多表达式，这类似于树的分支重复分裂的方式。



我们将这个解析器与我们第 9 章中编写的配置文件格式解析器进行对比，第 9 章中的解析器结构很简单：将输入文件划分成行，并逐行处理。而且每一行只有几种简单的语法形式。

我们必须使用不同方法来解决这里的问题。Egg 中并没有表达式按行分隔，而且表达式之间还有递归结构。应用表达式包含其他表达式。

所幸我们可以使用递归的方式编写一个解析器函数，并优雅地解决该问题，这反映了语言自身就是递归的。

我们定义了一个函数`parseExpression`，该函数接受一个字符串，并返回一个对象，包含了字符串起始位置处的表达式与解析表达式后剩余的字符串。当解析子表达式时（比如应用的参数），可以再次调用该函数，返回参数表达式和剩余字符串。剩余的字符串可以包含更多参数，也有可以是一个表示参数列表结束的右括号。

这里给出部分解析器代码。

```
function parseExpression(program) {
  program = skipSpace(program);
  let match, expr;
  if (match = /^"([^"]*)"/.exec(program)) {
    expr = {type: "value", value: match[1]};
  } else if (match = /^\d+\b/.exec(program)) {
    expr = {type: "value", value: Number(match[0])};
  } else if (match = /^[^\s(),"]+/.exec(program)) {
    expr = {type: "word", name: match[0]};
  } else {
    throw new SyntaxError("Unexpected syntax: " + program);
  }

  return parseApply(expr, program.slice(match[0].length));
}

function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}
```

由于 Egg 和 JavaScript 一样，允许其元素之间有任意数量的空白，所以我们必须在程序字符串的开始处重复删除空白。 这就是`skipSpace`函数能提供的帮助。

跳过开头的所有空格后，`parseExpression`使用三个正则表达式来检测 Egg 支持的三种原子的元素：字符串、数值和单词。解析器根据不同的匹配结果构造不同的数据类型。如果这三种形式都无法与输入匹配，那么输入就是一个非法表达式，解析器就会抛出异常。我们使用`SyntaxError`而不是`Error`作为异常构造器，这是另一种标准错误类型，因为它更具体 - 它也是在尝试运行无效的 JavaScript 程序时，抛出的错误类型。

接下来，我们从程序字符串中删去匹配的部分，将剩余的字符串和表达式对象一起传递给`parseApply`函数。该函数检查表达式是否是一个应用，如果是应用则解析带括号的参数列表。

```
function parseApply(expr, program) {
  program = skipSpace(program);
  if (program[0] != "(") {
    return {expr: expr, rest: program};
  }

  program = skipSpace(program.slice(1));
  expr = {type: "apply", operator: expr, args: []};
  while (program[0] != ")") {
    let arg = parseExpression(program);
    expr.args.push(arg.expr);
    program = skipSpace(arg.rest);
    if (program[0] == ",") {
      program = skipSpace(program.slice(1));
    } else if (program[0] != ")") {
      throw new SyntaxError("Expected ',' or ')'");
    }
  }
  return parseApply(expr, program.slice(1));
}
```

如果程序中的下一个字符不是左圆括号，说明当前表达式不是一个应用，parseApply会返回该表达式。

否则，该函数跳过左圆括号，为应用表达式创建语法树。接着递归调用`parseExpression`解析每个参数，直到遇到右圆括号为止。此处通过`parseApply`和`parseExpression`互相调用，实现函数间接递归调用。

因为我们可以使用一个应用来操作另一个应用表达式（比如`multiplier(2)(1)`），所以`parseApply`解析完一个应用后必须再次调用自身检查是否还有另一对圆括号。

这就是我们解析 Egg 所需的全部代码。我们使用`parse`函数来包装`parseExpression`，在解析完表达式之后验证输入是否到达结尾（一个 Egg 程序是一个表达式），遇到输入结尾后会返回整个程序对应的数据结构。

```
function parse(program) {
  let {expr, rest} = parseExpression(program);
  if (skipSpace(result.rest).length > 0) {
    throw new SyntaxError("Unexpected text after program");
  }
  return expr;
}

console.log(parse("+(a, 10)"));
// → {type: "apply",
//    operator: {type: "word", name: "+"},
//    args: [{type: "word", name: "a"},
//           {type: "value", value: 10}]}
```

程序可以正常工作了！当表达式解析失败时，解析函数不会输出任何有用的信息，也不会存储出错的行号与列号，而这些信息都有助于之后的错误报告。但考虑到我们的目的，这门语言目前已经足够优秀了。

求值器（evaluator）

在有了一个程序的语法树之后，我们该做什么呢？当然是执行程序了！而这就是求值器的功能。我们将语法树和作用域对象传递给求值器，执行器就会求解语法树中的表达式，然后返回整个过程的结果。

```
const specialForms = Object.create(null);

function evaluate(expr, scope) {
  if (expr.type == "value") {
    return expr.value;
  } else if (expr.type == "word") {
    if (expr.name in scope) {
      return scope[expr.name];
    } else {
      throw new ReferenceError(
        `Undefined binding: ${expr.name}`);
    }
  } else if (expr.type == "apply") {
    let {operator, args} = expr;
    if (operator.type == "word" &&
        operator.name in specialForms) {
      return specialForms[operator.name](expr.args, scope);
    } else {
      let op = evaluate(operator, scope);
      if (typeof op == "function") {
        return op(...args.map(arg => evaluate(arg, scope)));
      } else {
        throw new TypeError("Applying a non-function.");
      }
    }
  }
}
```

求值器为每一种表达式类型都提供了相应的处理逻辑。字面值表达式产生自身的值（例如，表达式 `100` 的求值为数值 `100` ）。对于绑定而言，我们必须检查程序中是否实际定义了该绑定，如果已经定义，则获取绑定的值。

应用则更为复杂。若应用有特殊形式（比如`if`），我们不会求解任何表达式，而是将表达式参数和环境传递给处理这种形式的函数。如果是普通调用，我们求解运算符，验证其是否是函数，并使用求值后的参数调用函数。

我们使用一般的 JavaScript 函数来表示 Egg 的函数。在定义特殊格式`fun`时，我们再回过头来看这个问题。

`evaluate`的递归结构类似于解析器的结构。两者都反映了语言自身的结构。我们也可以将解析器和求值器集成到一起，在解析的同时求解表达式，但将其分离为两个阶段使得程序更易于理解。

这就是解释 Egg 所需的全部代码。这段代码非常简单，但如果不定义一些特殊的格式，或向环境中添加一些有用的值，你无法使用该语言完成很多工作。

特殊形式

`specialForms`对象用于定义 Egg 中的特殊语法。该对象将单词和求解这种形式的函数关联起来。目前该对象为空，现在让我们添加`if`。

```
specialForms.if = (args, scope) => {
  if (args.length != 3) {
    throw new SyntaxError("Wrong number of args to if");
  } else if (evaluate(args[0], scope) !== false) {
    return evaluate(args[1], scope);
  } else {
    return evaluate(args[2], scope);
  }
};
```

Egg 的`if`语句需要三个参数。Egg 会求解第一个参数，若结果不是`false`，则求解第二个参数，否则求解第三个参数。相较于 JavaScript 中的`if`语句，Egg 的`if`形式更类似于 JavaScript 中的`?:`运算符。这是一条表达式，而非语句，它会产生一个值，即第二个或第三个参数的结果。

Egg 和 JavaScript 在处理条件值时也有些差异。Egg 不会将 0 或空字符串作为假，只有当值确实为`false`时，测试结果才为假。

我们之所以需要将`if`表达为特殊形式，而非普通函数，是因为函数的所有参数需要在函数调用前求值完毕，而`if`则只应该根据第一个参数的值，确定求解第二个还是第三个参数。`while`的形式也是类似的。

```
specialForms.while = (args, scope) => {
  if (args.length != 2) {
    throw new SyntaxError("Wrong number of args to while");
  }
  while (evaluate(args[0], scope) !== false) {
    evaluate(args[1], scope);
  }

  // Since undefined does not exist in Egg, we return false,
  // for lack of a meaningful result.
  return false;
};
```

另一个基本的积木是`do`，会自顶向下执行其所有参数。整个`do`表达式的值是最后一个参数的值。

```
specialForms.do = (args, scope) => {
  let value = false;
  for (let arg of args) {
    value = evaluate(arg, scope);
  }
};
```

我们还需要创建名为`define`的形式，来创建绑定对绑定赋值。`define`的第一个参数是一个单词，第二个参数是一个会产生值的表达式，并将第二个参数的计算结果赋值给第一个参数。由于`define`也是个表达式，因此必须返回一个值。我们则规定`define`应该将我们赋予绑定的值返回（就像 JavaScript 中的`=`运算符一样）。

```
specialForms.define = (args, scope) => {
  if (args.length != 2 || args[0].type != "word") {
    throw new SyntaxError("Incorrect use of define");
  }
  let value = evaluate(args[1], scope);
  scope[args[0].name] = value;
  return value;
};
```

环境

`evaluate`所接受的作用域是一个对象，它的名称对应绑定名称，它的值对应这些绑定所绑定的值。 我们定义一个对象来表示全局作用域。

我们需要先定义布尔绑定才能使用之前定义的`if`语句。由于只有两个布尔值，因此我们不需要为其定义特殊语法。我们简单地将`true`、`false`两个名称与其值绑定即可。

```
const topEnv = Object.create(null);

topScope.true = true;
topScope.false = false;
```

我们现在可以求解一个简单的表达式来对布尔值求反。

```
let prog = parse(`if(true, false, true)`);
console.log(evaluate(prog, topScope));
// → false
```

为了提供基本的算术和比较运算符，我们也添加一些函数值到作用域中。为了确保代码短小，我们在循环中使用`Function`来合成一批运算符，而不是分别定义所有运算符。

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {
  topScope[op] = Function("a, b", `return a ${op} b;`);
}
```

输出也是一个实用的功能，因此我们将`console.log`包装在一个函数中，并称之为`print`。

```
topScope.print = value => {
  console.log(value);
  return value;
};
```

这样一来我们就有足够的基本工具来编写简单的程序了。下面的函数提供了一个便利的方式来编写并运行程序。它创建一个新的环境对象，并解析执行我们赋予它的单个程序。

```
function run(program) {
  return evaluate(parse(program), Object.create(topScope));
}
```

我们将使用对象原型链来表示嵌套的作用域，以便程序可以在不改变顶级作用域的情况下，向其局部作用域添加绑定。

```
run(`
do(define(total, 0),
   define(count, 1),
   while(<(count, 11),
         do(define(total, +(total, count)),
            define(count, +(count, 1)))),
   print(total))
`);
// → 55
```

我们之前已经多次看到过这个程序，该程序计算数字 1 到 10 的和，只不过这里使用 Egg 语言表达。很明显，相较于实现同样功能的 JavaScript 代码，这个程序并不优雅，但对于一个不足 150 行代码的程序来说已经很不错了。

函数

每个功能强大的编程语言都应该具有函数这个特性。

幸运的是我们可以很容易地添加一个`fun`语言构造，`fun`将最后一个参数当作函数体，将之前的所有名称用作函数参数。

```
specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("Functions need a body");
  let body = args[args.length - 1];
  let params = args.slice(0, args.length - 1).map(expr => {
    if (expr.type != "word") {
      throw new SyntaxError("Parameter names must be words");
    }
    return expr.name;
  });

  return function() {
    if (arguments.length != argNames.length) {
      throw new TypeError("Wrong number of arguments");
    }
    let localScope = Object.create(scope);
    for (let i = 0; i < arguments.length; i++) {
      localScope[params[i]] = arguments[i];
    }
    return evaluate(body, localScope);
  };
};
```

Egg 中的函数可以获得它们自己的局部作用域。 `fun` 形式产生的函数创建这个局部作用域，并将参数绑定添加到它。 然后求解此范围内的函数体并返回结果。

```
run(`
do(define(plusOne, fun(a, +(a, 1))),
   print(plusOne(10)))
`);
// → 11

run(`
do(define(pow, fun(base, exp,
     if(==(exp, 0),
        1,
        *(base, pow(base, -(exp, 1)))))),
   print(pow(2, 10)))
`);
// → 1024
```

编译

我们构建的是一个解释器。在求值期间，解释器直接作用域由解析器产生的程序的表示。

编译是在解析和运行程序之间添加的另一个步骤：通过事先完成尽可能多的工作，将程序转换成一些可以高效求值的东西。例如，在设计良好的编程语言中，使用每个绑定时绑定引用的内存地址都是明确的，而不需要在程序运行时进行动态计算。这样可以省去每次访问绑定时搜索绑定的时间，只需要直接去预先定义好的内存位置获取绑定即可。

一般情况下，编译会将程序转换成机器码（计算机处理可以执行的原始格式）。但一些将程序转换成不同表现形式的过程也被认为是编译。

我们可以为 Egg 编写一个可供选择的求值策略，首先使用 `Function`，调用 JavaScript 编译器编译代码，将 Egg 程序转换成 JavaScript 程序，接着执行编译结果。若能正确实现该功能，可以使得 Egg 运行的非常快，而且实现这种编译器确实非常简单。

如果读者对该话题感兴趣，愿意花费一些时间在这上面，建议你尝试实现一个编译器作为练习。

站在别人的肩膀上

我们定义`if`和`while`的时候，你可能注意到他们封装得或多或少和 JavaScript 自身的`if`、`while`有点像。同样的，Egg 中的值也就是 JavaScript 中的值。

如果读者比较一下两种 Egg 的实现方式，一种是基于 JavaScrip t之上，另一种是直接使用机器提供的功能构建程序设计语言，会发现第二种方案需要大量工作才能完成，而且非常复杂。不管怎么说，本章的内容就是想让读者对编程语言的运行方式有一个基本的了解。

当需要完成一些任务时，相比于自己完成所有工作，借助于别人提供的功能是一种更高效的方式。虽然在本章中我们编写的语言就像玩具一样，十分简单，而且无论在什么情况下这门语言都无法与 JavaScript 相提并论。但在某些应用场景中，编写一门微型语言可以帮助我们更好地完成工作。

这些语言不需要像传统的程序设计语言。例如，若 JavaScript 没有正则表达式，你可以为正则表达式编写自己的解析器和求值器。

或者想象一下你在构建一个巨大的机械恐龙，需要编程实现恐龙的行为。JavaScript 可能不是实现该功能的最高效方式，你可以选择一种语言作为替代，如下所示：

```
behavior walk
  perform when
    destination ahead
  actions
    move left-foot
    move right-foot

behavior attack
  perform when
    Godzilla in-view
  actions
    fire laser-eyes
    launch arm-rockets
```

这通常被称为领域特定语言（Domain-specific Language），一种为表达极为有限的知识领域而量身定制的语言。它可以准确描述其领域中需要表达的事物，而没有多余元素。这种语言比通用语言更具表现力。

习题

数组

在 Egg 中支持数组需要将以下三个函数添加到顶级作用域：`array(...values)` 用于构造一个包含参数值的数组，`length(array)` 用于获取数组长度，`element(array, n)` 用于获取数组中的第 `n` 个元素。

```
// Modify these definitions...

topEnv.array = "...";

topEnv.length = "...";

topEnv.element = "...";

run(`
do(define(sum, fun(array,
     do(define(i, 0),
        define(sum, 0),
        while(<(i, length(array)),
          do(define(sum, +(sum, element(array, i))),
             define(i, +(i, 1)))),
        sum))),
   print(sum(array(1, 2, 3))))
`);
// → 6
```

闭包

我们定义 `fun` 的方式允许函数引用其周围环境，就像 JavaScript 函数一样，函数体可以使用在定义该函数时可以访问的所有局部绑定。

下面的程序展示了该特性：函数f返回一个函数，该函数将其参数和f的参数相加，这意味着为了使用绑定a，该函数需要能够访问f中的局部作用域。

```
run(`
do(define(f, fun(a, fun(b, +(a, b)))),
   print(f(4)(5)))
`);
// → 9
```

回顾一下fun形式的定义，解释一下该机制的工作原理。

注释

如果我们可以在 Egg 中编写注释就太好了。例如，无论何时，只要出现了井号（`#`），我们都将该行剩余部分当成注释，并忽略之，就类似于 JavaScript 中的 `//`。

解析器并不需要为支持该特性进行大幅修改。我们只需要修改 `skipSpace` 来像跳过空白符号一样跳过注释即可，此时调用 `skipSpace` 时不仅会跳过空白符号，还会跳过注释。修改代码，实现这样的功能。

```
// This is the old skipSpace. Modify it...
function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}

console.log(parse("# hello\nx"));
// → {type: "word", name: "x"}

console.log(parse("a # one\n   # two\n()"));
// → {type: "apply",
//    operator: {type: "word", name: "a"},
//    args: []}
```

修复作用域

目前绑定赋值的唯一方法是`define`。该语言构造可以同时实现定义绑定和将一个新的值赋予已存在的绑定。

这种歧义性引发了一个问题。当你尝试为一个非局部绑定赋予新值时，你最后会定义一个局部绑定并替换掉原来的同名绑定。一些语言的工作方式正和这种设计一样，但是我总是认为这是一种笨拙的作用域处理方式。

添加一个类似于`define`的特殊形式`set`，该语句会赋予一个绑定新值，若绑定不存在于内部作用域，则更新其外部作用域相应绑定的值。若绑定没有定义，则抛出`ReferenceError`（另一个标准错误类型）。

我们目前采取的技术是使用简单的对象来表示作用域对象，处理目前的任务非常方便，此时我们需要更进一步。你可以使用`Object.getPrototypeOf`函数来获取对象原型。同时也要记住，我们的作用域对象并未继承`Object.prototype`，因此若想调用`hasOwnProperty`，需要使用下面这个略显复杂的表达式。

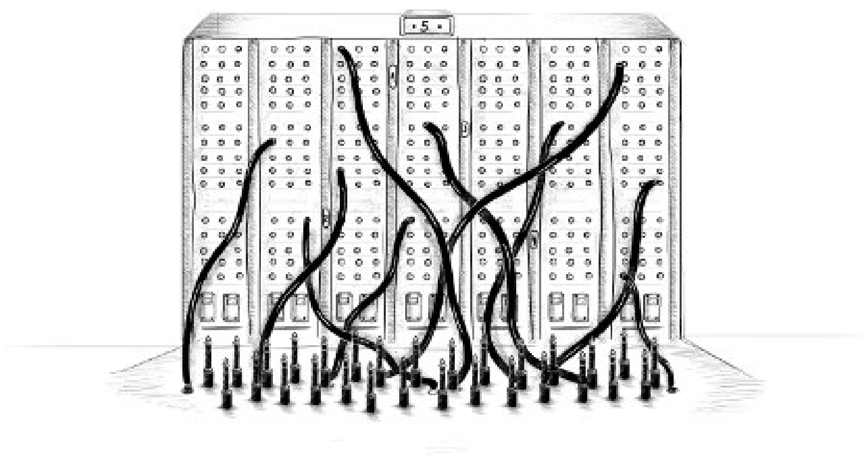
```
Object.prototype.hasOwnProperty.call(scope, name);
```

```
specialForms.set = function(args, env) {
  // Your code here.
};

run(`
do(define(x, 4),
   define(setx, fun(val, set(x, val))),
   setx(50),
   print(x))
`);
// → 50
run(`set(quux, true)`);
// → Some kind of ReferenceError
```

十三、浏览器中的 JavaScript

> 原文：JavaScript and the Browser
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> Web 背后的梦想是公共信息空间，其中我们通过共享信息进行交流。 其普遍性至关重要：超文本链接可指向任何东西，无论是个人的，本地的还是全球的，无论是草稿还是高度润色的。
>
> Douglas Crockford，《JavaScript 编程语言》（视频讲座）



本书接下来的章节将会介绍 Web 浏览器。可以说，没有浏览器，就没有 JavaScript。就算有，估计也不会有多少人去关心这门编程语言。

Web 技术自出现伊始，其演变方式和技术上就是以分散的方式发展的。许多浏览器厂商专门为其开发新的功能，有时这些新功能被大众采纳，有时这些功能被其他功能所代替，最终形成了一套标准。

这种发展模式是把双刃剑。一方面，不会有一个集中式的组织来管理技术的演进，取而代之的是一个包含多方利益集团的松散协作架构（偶尔会出现对立）。另一方面，互联网这种无计划的发展方式所开发出来的系统，其内部很难实现一致性。事实上，它的一些部分令人疑惑，并且毫无设计。

网络和 Internet

计算机网络出现在 20 世纪 50 年代。如果在两台或多台计算机之间铺设电缆，那么你可以通过这些电缆互相收发数据，并实现一些神奇的功能。

如果通过连接同一个建筑中的两台机器就可以实现一些神奇的功能，那么如果可以连接全世界的机器，就可以完成更伟大的工作了。20 世纪 80 年代，人们开发了相关技术来实现这个愿景，我们将其产生的网络称为 Internet。而 Internet 的表现名副其实。

计算机可以使用这种网络向其他计算机发送位数据。为了在传输位数据的基础上，实现计算机之间的有效通信，网络两端的机器必须知道这些位所表达的实际含义。对于给定的位序列，其含义完全取决于位序列描述的信息类型与使用的编码机制。

网络协议描述了一种网络通信方式。网络协议非常多，其中包括邮件发送、邮件收取和邮件共享，甚至连病毒软件感染控制计算机都有相应的协议。

例如，HTTP（超文本传输协议，Hypertext Transfer Protocol）是用于检索命名资源（信息块，如网页或图片）的协议。 它指定发出请求的一方应该以这样的一行开始，命名资源和它正在尝试使用的协议的版本。

```
GET /index.html HTTP/1.1
```

有很多规则，关于请求者在请求中包含更多信息的方式，以及另一方返回资源并打包其内容的方式。 我们将在第 18 章中更详细地观察 HTTP。

大多数协议都建立在其他协议之上。 HTTP 将网络视为一种流式设备，您可以将位放入这些设备，并使其按正确的顺序到达正确的目的地。 我们在第 11 章]中看到，确保这些事情已经是一个相当困难的问题。

TCP（传输控制协议，Transmission Control Protocol）就可以帮助我们解决该问题。所有连接到互联网的设备都会使用到这种协议，而多数互联网通信都构建在这种协议之上。

TCP 连接的工作方式是一台电脑必须等待或者监听，而另一台电脑则开始与之通信。一台机器为了同时监听不同类型的通信信息，会为每个监听器分配一个与之关联的数字（我们称之为端口）。大多数协议都指定了默认使用的端口。例如，当我们向使用 SMTP 协议发送一封邮件时，我们需要通过一台机器来发送邮件，而发送邮件的机器需要监听端口 25。

随后另一台机器连接到使用了正确端口号的目标机器上。如果可以连接到目标机器，而且目标机器在监听对应端口，则说明连接创建成功。负责监听的计算机名为服务器，而连接服务器的计算机名为客户端。

我们可以将该连接看成双向管道，位可以在其中流动，也就是说两端的机器都可以向连接中写入数据。当成功传输完这些位数据后，双方都可以读取另一端传来的数据。TCP 是一个非常便利的模型。我们可以说TCP就是一种网络的抽象。

Web

万维网（World Wide Web，不要将其与 Internet 混淆）是包含一系列协议和格式的集合，允许我们通过浏览器访问网页。词组中的 Web 指的是这些页面可以轻松地链接其他网页，因此最后可以连接成一张巨大的网，用户可以在网络中浏览。

你只需将一台计算机连接到 Internet 并使用 HTTP 监听 80 端口，就可以成为 Web 的一部分。其他计算机可以通过网络，并使用 HTTP 协议获取其他计算机上的文件。

网络中的每个文件都能通过 URL（统一资源定位符，Universal Resource Locator）访问，如下所示：

```
  http://eloquentjavascript.net/13_browser.html
 |      |                      |               |
 protocol       server               path
```

该地址的第一部分告诉我们 URL 使用的是 HTTP 协议（加密的 HTTP 连接则使用 `https://` 来表示）。第二部分指的是获取文件的服务器地址。第三部分是我们想要获取的具体文件（或资源）的路径。

连接到互联网的机器获得一个 IP 地址，该地址是一个数字，可用于将消息发送到该机器的，类似于 `"149.210.142.219"` 或 `"2001:4860:4860::8888"` 。但是或多或少的随机数字列表很难记住，而且输入起来很笨拙，所以你可以为一个特定的地址或一组地址注册一个域名。我注册了 `eloquentjavascript.net` ，来指向我控制的机器的 IP 地址，因此可以使用该域名来提供网页。

如果你在浏览器地址栏中输入上面提到的 URL，浏览器会尝试获取并显示该 URL 对应的文档。首先，你的浏览器需要找出域名 `eloquentjavascript.net` 指向的地址。然后使用 HTTP 协议，连接到该地址处的服务器，并请求 `/13_browser.html` 这个资源。如果一切顺利，服务器会发回一个文档，然后您的浏览器将显示在屏幕上。

HTML

HTML，即超文本标记语言（Hypertext Markup Language），是在网页中得到广泛使用的文档格式。HTML 文档不仅包含文本，还包含了标签，用于说明文本结构，描述了诸如链接、段落、标题之类的元素。

一个简短的 HTML 文档如下所示：

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

标签包裹在尖括号之间（ < 和 > ，小于和大于号），提供关于文档结构的信息。其他文本则是纯文本。

文档以 <!doctype html> 开头，告诉浏览器将页面解释为现代 HTML，以别于过去使用的各种方言。

HTML 文档有头部（head）和主体（body）。头部包含了文档信息，而主体则包含文档自身。在本例中，头部将文档标题声明为 "My home page" ，并使用 UTF-8 编码，它是将 Unicode 文本编码为二进制的方式。文档的主体包含标题（ <h1> ，表示一级标题， <h2> 到 <h6> 可以产生不同等级的子标题）和两个段落（ <p> ）。

标签有几种形式。一个元素，比如主体、段落或链接以一个起始标签（比如 <p> ）开始，并以一个闭合标签（比如 </p> ）结束。一些起始标签，比如一个链接（ <a> ），会包含一些额外信息，其形式是 name="value" 这种键值对，我们称之为属性。在本例中，使用属性 href="http://eloquentjavascript.net" 指定链接的目标，其中 href 表示“超文本链接（Hypertext Reference）”。

某些类型的标签不会包含任何元素，这种标签不需要闭合。元数据标签 <meta charset="utf-8"> 就是一个例子。

> 译者注：最好还是这样闭合它们： <meta charset="utf-8" /> 。

尽管 HTML 中尖括号有特殊含义，但为了在文档的文本中包含这些符号，可以引入另外一种形式的特殊标记方法。普通文本中的起始尖括号写成 < （less than），而闭合尖括号写成 > （greater than）。在 HTML 中，我们将一个 & 字符后跟着一个单词和分号（ ; ）这种写法称为一个实体，浏览器会使用实体编码对应的字符替换它们。

与之类似的是 JavaScript 字符串中反斜杠的使用。由于 HTML 中的实体机制赋予了 & 特殊含义，因此我们需要使用 & 来表示一个 & 字符。在属性的值（包在双引号中）中使用 " 可以插入实际的引号字符。

HTML 的解析过程容错性非常强。当应有的标签丢失时，浏览器会重新构建这些标签。标签的重新构建已经标准化，你可以认为所有现代浏览器的行为都是一致的。

下面的文件与之前版本显示效果相同：

```
<!doctype html>

<meta charset=utf-8>
<title>My home page</title>

<h1>My home page</h1>
<p>Hello, I am Marijn and this is my home page.
<p>I also wrote a book! Read it
  <a href=http://eloquentjavascript.net>here</a>.
```

`<html>`、`<head>`和`<body>`标签可以完全丢弃。浏览器知道`<meta>`和`<title>`属于头部，而`<h1>`属于主体。此外，我再也不用明确关闭某个段落，因为新段落开始或文档结束时，浏览器会隐式关闭段落标签。目标链接两边的引号也可以丢弃。

本书的示例通常都会省略`<html>`、`<head>`和`<body>`标签，以保持源代码简短，避免太过杂乱。但我会明确关闭所有标签并在属性两旁包含引号。

本书也会经常忽略`doctype`和`charset`声明。这并不是鼓励大家省略它们。当你忘记它们时，浏览器往往会做出荒谬的事情。 您应该认为`doctype`和`charset`元数据隐式出现在示例中，即使它们没有实际显示在文本中。

HTML 和 JavaScript

对于本书来说，最重要的一个 HTML 标签是`<script>`。该标签允许我们在文档中包含一段 JavaScript 代码。

```
<h1>Testing alert</h1>
<script>alert("hello!");</script>
```

当浏览器在读取 HTML 时，一旦遇到`<script>`标签就会执行该代码。这个页面在打开时会弹出一个对话框 - `alert`函数类似`prompt`，因为它弹出一个小窗口，但只显示一条消息而不请求输入。

在 HTML 文档中包含大程序是不切实际的。`<script>`标签可以指定一个`src`属性，从一个 URL 获取脚本文件（包含 JavaScript 程序的文本文件）。

```
<h1>Testing alert</h1>
<script src="code/hello.js"></script>
```

这里包含的文件`code/hello.js`是和上文中相同的一段程序，`alert("hello")`。当一个页面将其他 URL 引用为自身的一部分时（比如图像文件或脚本），网页浏览器将会立即获取这些资源并将其包含在页面中。

即使`script`标签引用了一个文本文件，且并未包含任何代码，你也必须使用`</script>`来闭合标签。如果你忘记了这点，浏览器会将剩余的页面会作为脚本的一部分进行解析。

你可以在浏览器中加载ES模块（参见第 10 章），向脚本标签提供 `type ="module"` 属性。 这些模块可以依赖于其他模块，通过将相对于自己的 URL 用作 `import` 声明中的模块名称。

某些属性也可以包含 JavaScript 程序。下面展示的 `<button>` 标签（显示一个按钮）有一个 `onclick` 属性。该属性的值将在点击按钮时运行。

```
<button onclick="alert('Boom!');">DO NOT PRESS</button>
```

需要注意的是，我们在 `onclick` 属性的字符串中使用了单引号，这是因为我们在使用了双引号来引用整个属性。我们也可以使用 `"` 。

沙箱

直接执行从因特网中下载的程序存在潜在危险。你不了解大多数的网页开发者，他们不一定都心怀善意。一旦运行某些不怀好意的人提供的程序，你的电脑可能会感染病毒，这些程序还会窃取数据会并盗走账号。

但网络的吸引力就在于你可以浏览网站，而不必要信任所有网站。这就是为什么浏览器严重限制了 JavaScript 程序的能力—— JavaScript 无法查看电脑中的任何文件，也无法修改与其所在页面无关的数据。

我们将这种隔离程序运行环境的技术称为沙箱。以该思想编写的程序在沙箱中运行，不会对计算机造成任何伤害。但是你应该想象，这种特殊的沙箱上面有一个厚钢筋笼子，所以在其中运行的程序实际上不会出去。

实现沙箱的难点是：一方面我们要给予程序一定的自由使得程序能有实际用处，但又要限制程序，防止其执行危险的行为。许多实用功能（比如与服务器通信或从剪贴板读取内容）也会存在问题，有些侵入者可以利用这些功能来侵入你的计算机。

时不时会有一些人想到新方法，突破浏览器的限制，并对你的机器造成伤害，从窃取少量的私人信息到掌握执行浏览器的整个机器。浏览器开发者的对策是修补漏洞，然后一切都恢复正常。直到下一次问题被发现并广为传播之前，某些政府或秘密组织可以私下利用这些漏洞。

兼容性与浏览器之争

在 Web 技术发展的早期，一款名为 Mosaic 的浏览器统治了整个市场。几年之后，这种平衡被 Netscape 公司打破，随后又被微软的 Internet Explorer 排挤出市场。无论什么时候，当一款浏览器统治了整个市场，浏览器供应商就会觉得他们有权利单方面为网络研发新的特性。由于大多数人都使用相同的浏览器，因此网站会开始使用这些独有特性，也就不再考虑其他浏览器的兼容性问题了。

这是兼容性的黑暗时代，我们通常称之为浏览器之争。网络开发者总是为缺乏统一的 Web 标准，而需要去考虑两到三种互不兼容的平台而感到烦恼。让事情变得更糟糕的是 2003 年左右使用的浏览器充满了漏洞，当然不同浏览器的漏洞都不一样。网页编写者的生活颇为艰辛。

Mozilla Firefox，作为 Netscape 浏览器的非盈利性分支，在20世纪初末期开始挑战 Internet Explorer 的霸主地位。因为当时微软并未特别关心与其竞争，导致 Firefox 迅速占领了很大的市场份额。与此同时，Google 发布了它的 Chrome 浏览器，而 Apple 的 Safari 也得到普及，导致现在成为四个主要选手的竞争，而非一家独大。

新的参与者对标准有着更认眞的态度，和更好的工程实践，为我们减少了不兼容性和错误。微软看到其市场份额极速下降，在其 Edge 浏览器中采取了这些态度，取代了 Internet Explorer。 如果您今天开始学习网络开发，请认为自己是幸运的。 主流浏览器的最新版本行为非常一致，并且错误相对较少。

这并不是说就没有问题了。某些使用网络的人，出于惰性或公司政策，被迫使用旧版本的浏览器。直到这些浏览器完全退出市场之前，为旧版本浏览器编写网站仍需要掌握很多不常见的特性，了解旧浏览器的缺陷和特殊之处。本书不会讨论这些特殊的特性，而着眼于介绍现代且健全的网络程序设计风格。

十四、文档对象模型

> 原文：The Document Object Model
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> Too bad! Same old story! Once you've finished building your house you notice you've accidentally learned something that you really should have known—before you started.
>
> Friedrich Nietzsche，《Beyond Good and Evil》



当你在浏览器中打开网页时，浏览器会接收网页的 HTML 文本并进行解析，其解析方式与第 11 章中介绍的解析器非常相似。浏览器构建文档结构的模型，并使用该模型在屏幕上绘制页面。

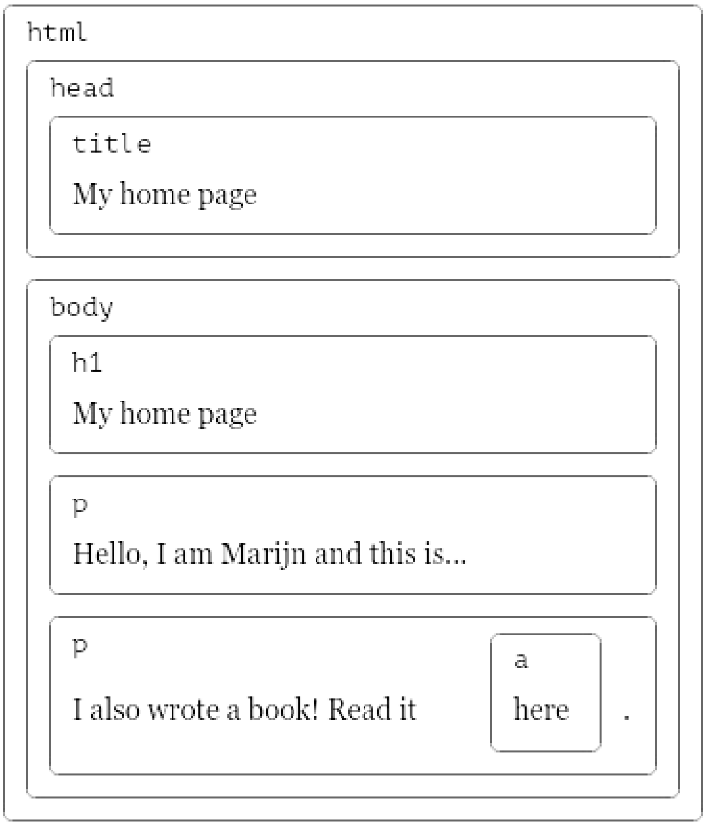
JavaScript 在其沙箱中提供了将文本转换成文档对象模型的功能。它是你可以读取或者修改的数据结构。模型是一个所见即所得的数据结构，改变模型会使得屏幕上的页面产生相应变化。

文档结构

你可以将 HTML 文件想象成一系列嵌套的箱子。诸如 `<body>` 和 `</body>` 之类的标签会将其他标签包围起来，而包含在内部的标签也可以包含其他的标签和文本。这里给出上一章中已经介绍过的示例文件。

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

该页面结构如下所示。



浏览器使用与该形状对应的数据结构来表示文档。每个盒子都是一个对象，我们可以和这些对象交互，找出其中包含的盒子与文本。我们将这种表示方式称为文档对象模型（Document Object Model），或简称 DOM。

我们可以通过全局绑定 `document` 来访问这些对象。该对象的 `documentElement` 属性引用了 `<html>` 标签对象。由于每个 HTML 文档都有一个头部和一个主体，它还具有 `head` 和 `body` 属性，指向这些元素。

树

回想一下第 12 章中提到的语法树。其结构与浏览器文档的结构极为相似。每个节点使用`children`引用其他节点，而每个子节点又有各自的`children`。其形状是一种典型的嵌套结构，每个元素可以包含与其自身相似的子元素。

如果一个数据结构有分支结构，而且没有任何环路（一个节点不能直接或间接包含自身），并且有一个单一、定义明确的“根节点”，那么我们将这种数据结构称之为树。就 DOM 来讲，`document.documentElement`就是其根节点。

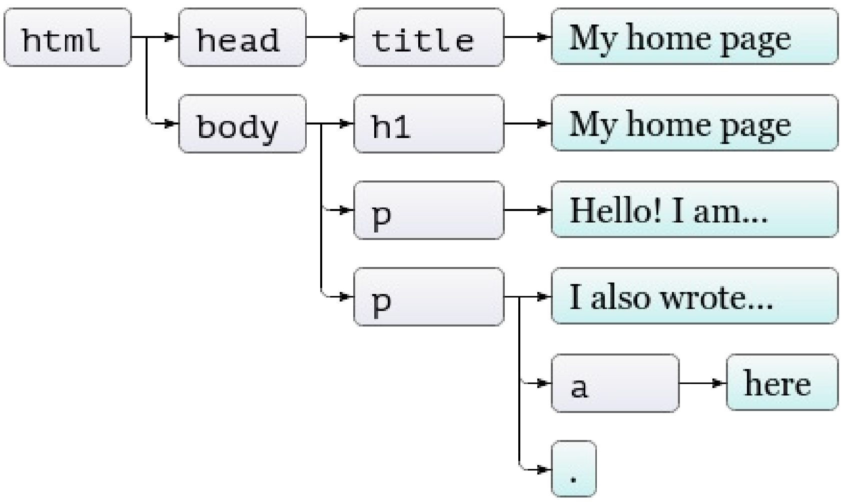
在计算机科学中，树的应用极为广泛。除了表现诸如 HTML 文档或程序之类的递归结构，树还可以用于维持数据的有序集合，因为在树中寻找或插入一个节点往往比在数组中更高效。

一棵典型的树有不同类型的节点。Egg 语言的语法树有标识符、值和应用节点。应用节点常常包含子节点，而标识符、值则是叶子节点，也就是没有子节点的节点。

DOM中也是一样。元素（表示 HTML 标签）的节点用于确定文档结构。这些节点可以包含子节点。这类节点中的一个例子是`document.body`。其中一些子节点可以是叶子节点，比如文本片段或注释。

每个 DOM 节点对象都包含`nodeType`属性，该属性包含一个标识节点类型的代码（数字）。元素的值为 1，DOM 也将该值定义成一个常量属性`document.ELEMENT_NODE`。文本节点（表示文档中的一段文本）代码为 3（`document.TEXT_NODE`）。注释的代码为 8（`document.COMMENT_NODE`）。

因此我们可以使用另一种方法来表示文档树：



叶子节点是文本节点，而箭头则指出了节点之间的父子关系。

标准

并非只有 JavaScript 会使用数字代码来表示节点类型。本章随后将会展示其他的 DOM 接口，你可能会觉得这些接口有些奇怪。这是因为 DOM 并不是为 JavaScript 而设计的，它尝试成为一组语言中立的接口，确保也可用于其他系统中，不只是 HTML，还有 XML。XML 是

一种通用数据格式，语法与 HTML 相近。

这就比较糟糕了。一般情况下标准都是非常易于使用的。但在这里其优势（跨语言的一致性）并不明显。相较于为不同语言提供类似的接口，如果能够将接口与开发者使用的语言进行适当集成，可以为开发者节省大量时间。

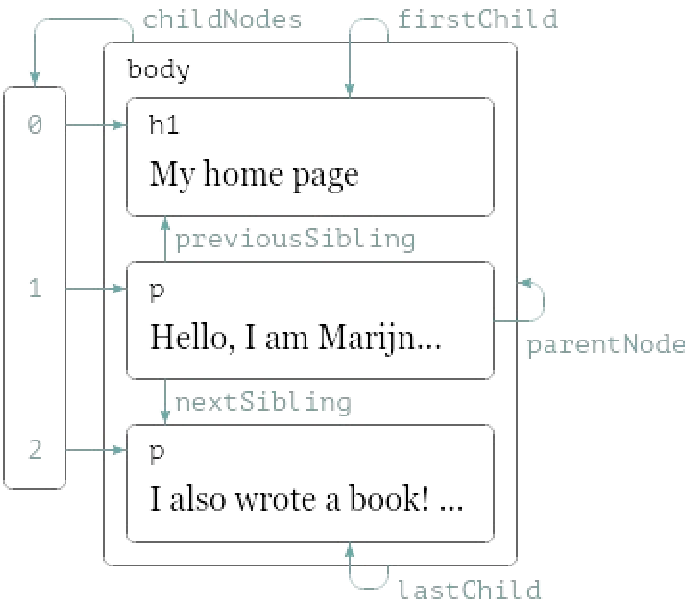
我们举例来说明一下集成问题。比如 DOM 中每个元素都有 `childNodes` 属性。该属性是一个类数组对象，有 `length` 属性，也可以使用数字标签访问对应的子节点。但该属性是 `NodeList` 类型的实例，而不是真正的数组，因此该类型没有诸如 `slice` 和 `map` 之类的方法。

有些问题是由不好的设计导致的。例如，我们无法在创建新的节点的同时立即为其添加子节点和属性。相反，你首先需要创建节点，然后使用副作用，将子节点和属性逐个添加到节点中。大量使用 DOM 的代码通常较长、重复和丑陋。

但这些问题并非无法改善。因为 JavaScript 允许我们构建自己的抽象，可以设计改进方式来表达你正在执行的操作。 许多用于浏览器编程的库都附带这些工具。

沿着树移动

DOM 节点包含了许多指向相邻节点的链接。下面的图表展示了这一点。



尽管图表中每种类型的节点只显示出一条链接，但每个节点都有 `parentNode` 属性，指向一个节点，它是这个节点的一部分。类似的，每个元素节点（节点类型为 1）均包含 `childNodes` 属性，该属性指向一个类数组对象，用于保存其子节点。

理论上，你可以通过父子之间的链接移动到树中的任何地方。但 JavaScript 也提供了一些更加方便的额外链接。 `firstChild` 属性和 `lastChild` 属性分别指向第一个子节点和最后一个子节点，若没有子节点则值为 `null` 。类似的， `previousSibling` 和 `nextSibling` 指向相邻节

点，分别指向拥有相同父亲的前一个节点和后一个节点。对于第一个子节点，`previousSibling`是`null`，而最后一个子节点的`nextSibling`则是`null`。

也存在`children`属性，它就像`childNodes`，但只包含元素（类型为 1）子节点，而不包含其他类型的子节点。当你对文本节点不感兴趣时，这可能很有用。

处理像这样的嵌套数据结构时，递归函数通常很有用。以下函数在文档中扫描包含给定字符串的文本节点，并在找到一个时返回`true`：

```
function talksAbout(node, string) {
  if (node.nodeType == document.ELEMENT_NODE) {
    for (let i = 0; i < node.childNodes.length; i++) {
      if (talksAbout(node.childNodes[i], string)) {
        return true;
      }
    }
    return false;
  } else if (node.nodeType == document.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}

console.log(talksAbout(document.body, "book"));
// → true
```

因为`childNodes`不是真正的数组，所以我们不能用`for/of`来遍历它，并且必须使用普通的`for`循环遍历索引范围。

文本节点的`nodeValue`属性保存它所表示的文本字符串。

查找元素

使用父节点、子节点和兄弟节点之间的连接遍历节点确实非常实用。但是如果我们只想查找文档中的特定节点，那么从`document.body`开始盲目沿着硬编码的链接路径查找节点并非良策。如果程序通过树结构定位节点，就需要依赖于文档的具体结构，而文档结构随后可能发生变化。另一个复杂的因素是 DOM 会为不同节点之间的空白字符创建对应的文本节点。例如示例文档中的`body`标签不止包含 3 个子节点（`<h1>`和两个`<p>`元素），其实包含 7 个子节点：这三个节点、三个节点前后的空格、以及元素之间的空格。

因此，如果你想获取文档中某个链接的`href`属性，最好不要去获取文档`body`元素中第六个子节点的第二个子节点，而最好直接获取文档中的第一个链接，而且这样的操作确实可以实现。

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

所有元素节点都包含`getElementsByTagName`方法，用于从所有后代节点中（直接或间接子节点）搜索包含给定标签名的节点，并返回一个类数组的对象。

你也可以使用`document.getElementById`来寻找包含特定`id`属性的某个节点。

```
<p>My ostrich Gertrude:</p>
<p><img id="gertrude" src="img/ostrich.png"></p>

<script>
  let ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

第三个类似的方法是`getElementsByClassName`，它与`getElementsByTagName`类似，会搜索元素节点的内容并获取所有包含特定`class`属性的元素。

修改文档

几乎所有 DOM 数据结构中的元素都可以被修改。文档树的形状可以通过改变父子关系来修改。 节点的`remove`方法将它们从当前父节点中移除。`appendChild`方法可以添加子节点，并将其放置在子节点列表末尾，而`insertBefore`则将第一个参数表示的节点插入到第二个参数表示的节点前面。

```
<p>One</p>
<p>Two</p>
<p>Three</p>

<script>
  let paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

每个节点只能存在于文档中的某一个位置。因此，如果将段落`Three`插入到段落`One`前，会将该节点从文档末尾移除并插入到文档前面，最后结果为`Three/One/Two`。所有将节点插入到某处的方法都有这种副作用——会将其从当前位置移除（如果存在的话）。

`replaceChild`方法用于将一个子节点替换为另一个子节点。该方法接受两个参数，第一个参数是新节点，第二个参数是待替换的节点。待替换的节点必须是该方法调用者的子节点。这里需要注意，`replaceChild`和`insertBefore`都将新节点作为第一个参数。

创建节点

假设我们要编写一个脚本，将文档中的所有图像（`<img>`标签）替换为其`alt`属性中的文本，该文本指定了图像的文字替代表示。

这不仅涉及删除图像，还涉及添加新的文本节点，并替换原有图像节点。为此我们使用`document.createTextNode`方法。

```html
<p>The <img src="img/cat.png" alt="Cat"> in the
  <img src="img/hat.png" alt="Hat">.</p>

<p><button onclick="replaceImages()">Replace</button></p>

<script>
  function replaceImages() {
    let images = document.body.getElementsByTagName("img");
    for (let i = images.length - 1; i >= 0; i--) {
      let image = images[i];
      var image = images[i];
      if (image.alt) {
        let text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>
```

给定一个字符串，`createTextNode`为我们提供了一个文本节点，我们可以将它插入到文档中，来使其显示在屏幕上。

该循环从列表末尾开始遍历图像。我们必须这样反向遍历列表，因为`getElementsByTagName`之类的方法返回的节点列表是动态变化的。该列表会随着文档改变还改变。若我们从列表头开始遍历，移除掉第一个图像会导致列表丢失其第一个元素，第二次循环时，因为集合的长度此时为 1，而`i`也为 1，所以循环会停止。

如果你想要获得一个固定的节点集合，可以使用数组的`Array.from`方法将其转换成实际数组。

```js
let arrayish = {0: "one", 1: "two", length: 2};
let array = Array.from(arrayish);
console.log(array.map(s => s.toUpperCase()));
// → ["ONE", "TWO"]
```

你可以使用`document.createElement`方法创建一个元素节点。该方法接受一个标签名，返回一个新的空节点，节点类型由标签名指定。

下面的示例定义了一个`elt`工具，用于创建一个新的元素节点，并将其剩余参数当作该节点的子节点。接着使用该函数为引用添加来源信息。

```
<blockquote id="quote">
  No book can ever be finished. While working on it we learn
  just enough to find it immature the moment we turn away
  from it.
</blockquote>

<script>
  function elt(type, ...children) {
    let node = document.createElement(type);
    for (let child of children) {
      if (typeof child != "string") node.appendChild(child);
      else node.appendChild(document.createTextNode(child));
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "—",
        elt("strong", "Karl Popper"),
        ", preface to the second editon of ",
        elt("em", "The Open Society and Its Enemies"),
        ", 1950"));
</script>
```

属性

我们可以通过元素的 DOM 对象的同名属性去访问元素的某些属性，比如链接的`href`属性。这仅限于最常用的标准属性。

HTML 允许你在节点上设定任何属性。这一特性非常有用，因为这样你就可以在文档中存储额外信息。你自己创建的属性不会出现在元素节点的属性中。你必须使用`getAttribute`和`setAttribute`方法来访问这些属性。

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>

<script>
  let paras = document.body.getElementsByTagName("p");
  for (let para of Array.from(paras)) {
    if (para.getAttribute("data-classified") == "secret") {
      para.remove();
    }
  }
</script>
```

建议为这些组合属性的名称添加`data-`前缀，来确保它们不与任何其他属性发生冲突。

这里有一个常用的属性：`class`。该属性是 JavaScript 中的保留字。因为某些历史原因（某些旧版本的 JavaScript 实现无法处理和关键字或保留字同名的属性），访问`class`的属性名为`className`。你也可以使用`getAttribute`和`setAttribute`方法，使用其实际名称`class`来访问该属性。

布局

你可能已经注意到不同类型的元素有不同的布局。某些元素，比如段落（`<p>`）和标题（`<h1>`）会占据整个文档的宽度，并且在独立的一行中渲染。这些元素被称为块（Block）元素。其他的元素，比如链接（`<a>` 或 `<strong>` 元素则与周围文本在同一行中渲染。这类元素我们称之为内联（Inline）元素。

对于任意特定文档，浏览器可以根据每个元素的类型和内容计算其尺寸与位置等布局信息。接着使用布局来绘制文档。

JavaScript 中可以访问元素的尺寸与位置。

属性 `offsetWidth` 和 `offsetHeight` 给出元素的起始位置（单位是像素）。像素是浏览器中的基本测量单元。它通常对应于屏幕可以绘制的最小的点，但是在现代显示器上，可以绘制非常小的点，这可能不再适用了，并且浏览器像素可能跨越多个显示点。

同样，`clientWidth` 和 `clientHeight` 向你提供元素内的空间大小，忽略边框宽度。

```
<p style="border: 3px solid red">
  I'm boxed in
</p>

<script>
  let para = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  console.log("offsetHeight:", para.offsetHeight);
</script>
```

`getBoundingClientRect` 方法是获取屏幕中某个元素精确位置的最有效方法。该方法返回一个对象，包含 `top` 、 `bottom` 、 `left` 和 `right` 四个属性，表示元素相对于屏幕左上角的位置（单位是像素）。若你想要知道其相对于整个文档的位置，必须加上其滚动位置，你可以在 `pageXOffset` 和 `pageYOffset` 绑定中找到。

我们还需要花些力气才能完成文档的排版工作。为了加快速度，每次你改变它时，浏览器引擎不会立即重新绘制整个文档，而是尽可能等待并推迟重绘操作。当一个修改文档的 JavaScript 程序结束时，浏览器会计算新的布局，并在屏幕上显示修改过的文档。若程序通过读取 `offsetHeight` 和 `getBoundingClientRect` 这类属性获取某些元素的位置或尺寸时，为了提供正确的信息，浏览器也需要计算布局。

如果程序反复读取 DOM 布局信息或修改 DOM，会强制引发大量布局计算，导致运行非常缓慢。下面的代码展示了一个示例。该示例包含两个不同的程序，使用 `X` 字符构建一条线，其长度是 2000 像素，并计算每个任务的时间。

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    let start = Date.now(); // Current time in milliseconds
    action();
    console.log(name, "took", Date.now() - start, "ms");
  }

  time("naive", () => {
    let target = document.getElementById("one");
    while (target.offsetWidth < 2000) {
      target.appendChild(document.createTextNode("X"));
    }
  });
  // → naive took 32 ms

  time("clever", function() {
    let target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (target.offsetWidth / 5));
    target.firstChild.nodeValue = "X".repeat(total);
  });
  // → clever took 1 ms
</script>
```

样式

我们看到了不同的 HTML 元素的绘制是不同的。一些元素显示为块，一些则是以内联方式显示。我们还可以添加一些样式，比如使用 `<strong>` 加粗内容，或使用 `<a>` 使内容变成蓝色，并添加下划线。

`<img>` 标签显示图片的方式或点击标签 `<a>` 时跳转的链接都和元素类型紧密相关。但元素的默认样式，比如文本的颜色、是否有下划线，都是可以改变的。这里给出使用 `style` 属性的示例。

```
<p><a href=".">Normal link</a></p>
<p><a href="." style="color: green">Green link</a></p>
```

样式属性可以包含一个或多个声明，格式为属性（比如 `color` ）后跟着一个冒号和一个值（比如 `green` ）。当包含更多声明时，不同属性之间必须使用分号分隔，比如 `color:red;border:none` 。

文档的很多方面会受到样式的影响。例如， `display` 属性控制一个元素是否显示为块元素或内联元素。

```
This text is displayed <strong>inline</strong>,
<strong style="display: block">as a block</strong>, and
<strong style="display: none">not at all</strong>.
```

`block` 标签会结束其所在的那一行，因为块元素是不会和周围文本内联显示的。最后一个标签完全不会显示出来，因为 `display:none` 会阻止一个元素呈现在屏幕上。这是隐藏元素的一种方式。更好的方式是将其从文档中完全移除，因为稍后将其放回去是一件很简单的事情。

JavaScript 代码可以通过元素的 `style` 属性操作元素的样式。该属性保存了一个对象，对象中存储了所有可能的样式属性，这些属性的值是字符串，我们可以把字符串写入属性，修改某些方面的元素样式。

```
<p id="para" style="color: purple">
  Nice text
</p>

<script>
  let para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

一些样式属性名包含破折号，比如 `font-family` 。由于这些属性的命名不适合在 JavaScript 中使用（你必须写成 `style["font-family"]` ），因此在 JavaScript 中，样式对象中的属性名都移除了破折号，并将破折号之后的字母大写（ `style.fontFamily` ）。

层叠样式

我们把 HTML 的样式化系统称为 CSS，即层叠样式表（Cascading Style Sheets）。样式表是一系列规则，指出如何为文档中元素添加样式。可以在 `<style>` 标签中写入 CSS。

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Now <strong>strong text</strong> is italic and gray.</p>
```

所谓层叠指的是将多条规则组合起来产生元素的最终样式。在示例中， `<strong>` 标签的默认样式 `font-weight:bold` ，会被 `<style>` 标签中的规则覆盖，并为 `<strong>` 标签样式添加 `font-style` 和 `color` 属性。

当多条规则重复定义同一属性时，最近的规则会拥有最高的优先级。因此如果 `<style>` 标签中的规则包含 `font-weight:normal` ，违背了默认的 `font-weight` 规则，那么文本将会显示为普通样式，而非粗体。属性 `style` 中的样式会直接作用于节点，而且往往拥有最高优先级。

我们可以在 CSS 规则中使用标签名来定位标签。规则 `.abc` 指的是所有 `class` 属性中包含 `abc` 的元素。规则 `#xyz` 作用于 `id` 属性为 `xyz` （应当在文档中唯一存在）的元素。

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* p elements with id main and with classes a and b */
p#main.a.b {
  margin-bottom: 20px;
}
```

优先级规则偏向于最近定义的规则，仅在规则特殊性相同时适用。规则的特殊性用于衡量该规则描述匹配元素时的准确性。特殊性取决于规则中的元素数量和类型（`tag`、`class`或`id`）。例如，目标规则`p.a`比目标规则`p`或`.a`更具体，因此有更高优先级。

`p>a`这种写法将样式作用于`<p>`标签的直系子节点。类似的，`p a`应用于所有的`<p>`标签中的`<a>`标签，无论是否是直系子节点。

查询选择器

本书不会使用太多样式表。尽管理解样式表对浏览器程序设计至关重要，想要正确解释所有浏览器支持的属性及其使用方式，可能需要两到三本书才行。

我介绍选择器语法（用在样式表中，确定样式作用的元素）的主要原因是这种微型语言同时也是一种高效的 DOM 元素查找方式。

`document`对象和元素节点中都定义了`querySelectorAll`方法，该方法接受一个选择器字符串并返回类数组对象，返回的对象中包含所有匹配的元素。

```
<p>And if you go chasing
  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="character">hookah smoking
  <span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>

<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));           // All <p> elements
  // → 4
  console.log(count(".animal"));     // Class animal
  // → 2
  console.log(count("p .animal"));   // Animal inside of <p>
  // → 2
  console.log(count("p > .animal")); // Direct child of <p>
  // → 1
</script>
```

与 `getElementsByTagName` 这类方法不同，由 `querySelectorAll` 返回的对象不是动态变更的。修改文档时其内容不会被修改。但它仍然不是一个真正的数组，所以如果你打算将其看做真的数组，你仍然需要调用 `Array.from`。

`querySelector` 方法（没有 `All`）与 `querySelectorAll` 作用相似。如果只想寻找某一个特殊元素，该方法非常有用。该方法只返回第一个匹配的元素，如果没有匹配的元素则返回 `null`。

位置与动画

`position` 样式属性是一种强大的布局方法。默认情况下，该属性值为 `static`，表示元素处于文档中的默认位置。若该属性设置为 `relative`，该元素在文档中依然占据空间，但此时其 `top` 和 `left` 样式属性则是相对于常规位置的偏移。若 `position` 设置为 `absolute`，会将元素从默认文档流中移除，该元素将不再占据空间，而会与其他元素重叠。其 `top` 和 `left` 属性则是相对其最近的闭合元素的偏移，其中 `position` 属性的值不是 `static`。如果没有任何闭合元素存在，则是相对于整个文档的偏移。

我们可以使用该属性创建一个动画。下面的文档用于显示一幅猫的图片，该图片会沿着椭圆轨迹移动。

```
<p style="text-align: center">
  <img src="img/cat.png" style="position: relative">
</p>
<script>
  let cat = document.querySelector("img");
  let angle = Math.PI / 2;
  function animate(time, lastTime) {
    if (lastTime != null) {
      angle += (time - lastTime) * 0.001;
    }
    lastTime = time;
    cat.style.top = (Math.sin(angle) * 20) + "px";
    cat.style.left = (Math.cos(angle) * 200) + "px";
    requestAnimationFrame(newTime => animate(newTime, time));
  }
  requestAnimationFrame(animate);
</script>
```

我们的图像在页面中央，`position` 为 `relative`。为了移动这只猫，我们需要不断更新图像的 `top` 和 `left` 样式。

脚本使用 `requestAnimationFrame` 在每次浏览器准备重绘屏幕时调用 `animate` 函数。`animate` 函数再次调用 `requestAnimationFrame` 以准备下一次更新。当浏览器窗口（或标签）激活时，更新频率大概为 60 次每秒，这种频率可以生成美观的动画。

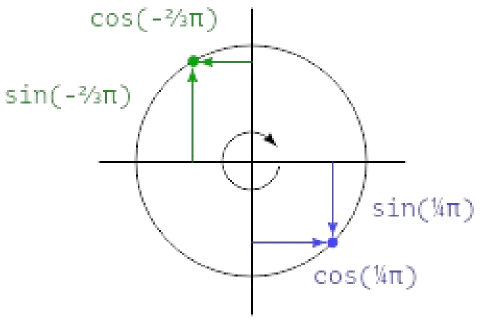
若我们只是在循环中更新 DOM，页面会静止不动，页面上也不会显示任何东西。浏览器不会在执行 JavaScript 程序时刷新显示内容，也不允许页面上的任何交互。这就是我们需要 `requestAnimationFrame` 的原因，该函数用于告知浏览器 JavaScript 程序目前已经完成工作，因此浏览器可以继续执行其他任务，比如刷新屏幕，响应用户动作。

我们将动画生成函数作为参数传递给 `requestAnimationFrame`。为了确保每一毫秒猫的移动是稳定的，而且动画是圆滑的，它基于一个速度，角度以这个速度改变这一次与上一次函数运行的差。如果仅仅每次走几步，猫的动作可能略显迟钝，例如，另一个在相同电脑上的繁重任务可能使得该函数零点几秒之后才会运行一次。

我们使用三角函数 `Math.cos` 和 `Math.sin` 来使猫沿着圆弧移动。你可能不太熟悉这些计算，我在这里简要介绍它们，因为你会在这本书中偶尔遇到。

`Math.cos` 和 `Math.sin` 非常实用，我们可以利用一个 1 个弧度，计算出以点 `(0,0` 为圆心的圆上特定点的位置。两个函数都将参数解释为圆上的一个位置，0 表示圆上最右侧那个点，一直逆时针递增到 `2π`（大概是 6.28），正好走过整个圆。`Math.cos` 可以计算出圆上某一点对应的 `x` 坐标，而 `Math.sin` 则计算出 `y` 坐标。超过 `2π` 或小于 0 的位置（或角度）都是合法的。因为弧度是循环重复的，`a+2π` 与 `a` 的角度相同。

用于测量角度的单位称为弧度 - 一个完整的圆弧是 `2π` 个弧度，类似于以角度度量时的 360 度。常量 `π` 在 JavaScript 中为 `Math.PI`。



猫的动画代码保存了一个名为 `angle` 的计数器，该绑定记录猫在圆上的角度，而且每当调用 `animate` 函数时，增加该计数器的值。我们接着使用这个角度来计算图像元素的当前位置。`top` 样式是 `Math.sin` 的结果乘以 20，表示圆中的垂直弧度。`left` 样式是 Math.cos 的结果乘以 `200`，因此圆的宽度大于其高度，导致最后猫会沿着椭圆轨迹移动。

这里需要注意的是样式的值一般需要指定单位。本例中，我们在数字后添加 `px` 来告知浏览器以像素为计算单位（而非厘米，`ems`，或其他单位）。我们很容易遗漏这个单位。如果我们没有为样式中的数字加上单位，浏览器最后会忽略掉该样式，除非数字是 0，在这种情况下使用什么单位，其结果都是一样的。

本章小结

JavaScript 程序可以通过名为 DOM 的数据结构，查看并修改浏览器中显示的文档。该数据结构描述了浏览器文档模型，而 JavaScript 程序可以通过修改该数据结构来修改浏览器展示的文档。

DOM 的组织就像树一样，DOM 根据文档结构来层次化地排布元素。描述元素的对象包含很多属性，比如 `parentNode` 和 `childNodes` 这两个属性可以用来遍历 DOM 树。

我们可以通过样式来改变文档的显示方式，可以直接在节点上附上样式，也可以编写匹配节点的规则。样式包含许多不同的属性，比如 `color` 和 `display` 。JavaScript 代码可以直接通过节点的 `style` 属性操作元素的样式。

习题

创建一张表

HTML 表格使用以下标签结构构建：

```
<table>
  <tr>
    <th>name</th>
    <th>height</th>
    <th>place</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
  </tr>
</table>
```

`<table>` 标签中，每一行包含一个 `<tr>` 标签。 `<tr>` 标签内部则是单元格元素，分为表头（ `<th>` ）和常规单元格（ `<td>` ）。

给定一个山的数据集，一个包含 `name` ， `height` 和 `place` 属性的对象数组，为枚举对象的表格生成 DOM 结构。 每个键应该有一列，每个对象有一行，外加一个顶部带有 `<th>` 元素的标题行，列出列名。

编写这个程序，以便通过获取数据中第一个对象的属性名称，从对象自动产生列。

将所得表格添加到 `id` 属性为 `"mountains"` 的元素，以便它在文档中可见。

当你完成后，将元素的 `style.textAlign` 属性设置为 `right` ，将包含数值的单元格右对齐。

```
<h1>Mountains</h1>

<div id="mountains"></div>

<script>
  const MOUNTAINS = [
    {name: "Kilimanjaro", height: 5895, place: "Tanzania"},
    {name: "Everest", height: 8848, place: "Nepal"},
    {name: "Mount Fuji", height: 3776, place: "Japan"},
    {name: "Vaalserberg", height: 323, place: "Netherlands"},
    {name: "Denali", height: 6168, place: "United States"},
    {name: "Popocatepetl", height: 5465, place: "Mexico"},
    {name: "Mont Blanc", height: 4808, place: "Italy/France"}
  ];

  // Your code here
</script>
```

通过标签名获取元素

`document.getElementsByTagName` 方法返回带有特定标签名称的所有子元素。实现该函数，这里注意是函数不是方法。该函数的参数是一个节点和字符串（标签名称），并返回一个数组，该数组包含所有带有特定标签名称的所有后代元素节点。

你可以使用 `nodeName` 属性从 DOM 元素中获取标签名称。但这里需要注意，使用 `tagName` 获取的标签名称是全大写形式。可以使用字符串的 `toLowerCase` 或 `toUpperCase` 来解决这个问题。

```
<h1>Heading with a <span>span</span> element.</h1>
<p>A paragraph with <span>one</span>, <span>two</span>
  spans.</p>

<script>
  function byTagName(node, tagName) {
    // Your code here.
  }

  console.log(byTagName(document.body, "h1").length);
  // → 1
  console.log(byTagName(document.body, "span").length);
  // → 3
  let para = document.querySelector("p");
  console.log(byTagName(para, "span").length);
  // → 2
</script>
```

猫的帽子

扩展一下之前定义的用来绘制猫的动画函数，让猫和它的帽子沿着椭圆形轨道边（帽子永远在猫的对面）移动。

你也可以尝试让帽子环绕着猫移动，或修改成其他有趣的动画。

为了便于定位多个对象，一个比较好的方法是使用绝对（`absolute`）定位。这就意味着 `top` 和 `left` 属性是相对于文档左上角的坐标。你可以简单地在坐标上加上一个固定数字，以避免出现负的坐标，它会使图像移出可见页面。

```
<style>body { min-height: 200px }</style>
<img src="img/cat.png" id="cat" style="position: absolute">
<img src="img/hat.png" id="hat" style="position: absolute">

<script>
  let cat = document.querySelector("#cat");
  let hat = document.querySelector("#hat");

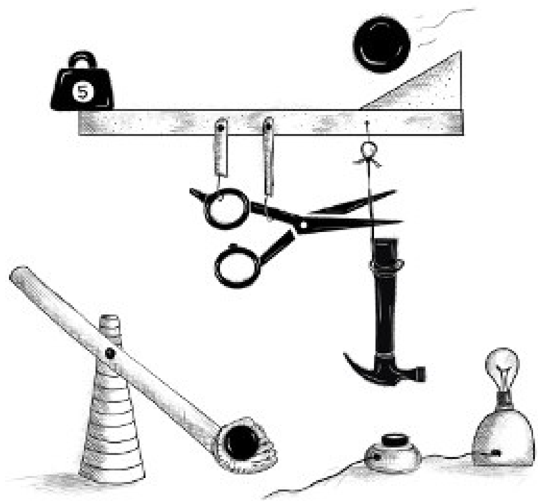
  let angle = 0;
  let lastTime = null;
  function animate(time) {
    if (lastTime != null) angle += (time - lastTime) * 0.001;
    lastTime = time;
    cat.style.top = (Math.sin(angle) * 40 + 40) + "px";
    cat.style.left = (Math.cos(angle) * 200 + 230) + "px";

    // Your extensions here.

    requestAnimationFrame(animate);
  }
 requestAnimationFrame(animate);
</script>
```

十五、处理事件

> 原文：Handling Events
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 你对你的大脑拥有控制权，而不是外部事件。认识到这一点，你就找到了力量。
>
> 马可·奥勒留，《沉思录》



有些程序处理用户的直接输入，比如鼠标和键盘动作。这种输入方式不是组织整齐的数据结构 - 它是一次一个地，实时地出现的，并且期望程序在发生时作出响应。

事件处理器

想象一下，有一个接口，若想知道键盘上是否有一个键是否被按下，唯一的方法是读取那个按键的当前状态。为了能够响应按键动作，你需要不断读取键盘状态，以在按键被释放之前捕捉到按下状态。这种方法在执行时间密集计算时非常危险，因为你可能错过按键事件。

一些原始机器可以像那样处理输入。有一种更进一步的方法，硬件或操作系统发现按键时间并将其放入队列中。程序可以周期性地检查队列，等待新事件并在发现事件时进行响应。

当然，程序必须记得监视队列，并经常做这种事，因为任何时候，按键被按下和程序发现事件之间都会使得软件反应迟钝。该方法被称为轮询。大多数程序员更希望避免这种方法。

一个更好的机制是，系统在发生事件时主动通知我们的代码。浏览器实现了这种特性，支持我们将函数注册为特定事件的处理器。

```
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

`window` 绑定指向浏览器提供的内置对象。它代表包含文档的浏览器窗口。调用它的 `addEventListener` 方法注册第二个参数，以便在第一个参数描述的事件发生时调用它。

事件与 DOM 节点

每个浏览器事件处理器被注册在上下文中。在为整个窗口注册处理器之前，我们在 `window` 对象上调用了 `addEventListener`。这种方法也可以在 DOM 元素和一些其他类型的对象上找到。仅当事件发生在其注册对象的上下文中时，才调用事件监听器。

```
<button>Click me</button>
<p>No handler here.</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Button clicked.");
  });
</script>
```

示例代码中将处理器附加到按钮节点上。因此，点击按钮时会触发并执行处理器，而点击文档的其他部分则没有反应。

向节点提供 `onclick` 属性也有类似效果。这适用于大多数类型的事件 - 您可以为属性附加处理器，属性名称为前面带有 `on` 的事件名称。

但是一个节点只能有一个 `onclick` 属性，所以你只能用这种方式为每个节点注册一个处理器。`addEventListener` 方法允许您添加任意数量的处理器，因此即使元素上已经存在另一个处理器，添加处理器也是安全的。

`removeEventListener` 方法将删除一个处理器，使用类似于 `addEventListener` 的参数调用。

```
<button>Act-once button</button>
<script>
  let button = document.querySelector("button");
  function once() {
    console.log("Done.");
    button.removeEventListener("click", once);
  }
  button.addEventListener("click", once);
</script>
```

赋予 `removeEventListener` 的函数必须是赋予 `addEventListener` 的完全相同的函数值。 因此，要注销一个处理其，您需要为该函数提供一个名称（在本例中为 `once` ），以便能够将相同的函数值传递给这两个方法。

事件对象

虽然目前为止我们忽略了它，事件处理器函数作为对象传递：事件（Event）对象。这个对象持有事件的额外信息。例如，如果我们想知道哪个鼠标按键被按下，我们可以查看事件对象的which属性。

```
<button>Click me any way you want</button>
<script>
  let button = document.querySelector("button");
  button.addEventListener("mousedown", event => {
    if (event.button == 0) {
      console.log("Left button");
    } else if (event.button == 1) {
      console.log("Middle button");
    } else if (event.button == 2) {
      console.log("Right button");
    }
  });
</script>
```

存储在各种类型事件对象中的信息是有差别的。随后本章将会讨论许多类型的事件。对象的 `type` 属性一般持有一个字符串，表示事件（例如 `"click"` 和 `"mousedown"` ）。

传播

对于大多数事件类型，在具有子节点的节点上注册的处理器，也将接收发生在子节点中的事件。若点击一个段落中的按钮，段落的事件处理器也会收到点击事件。

但若段落和按钮都有事件处理器，则先执行最特殊的事件处理器（按钮的事件处理器）。也就是说事件向外传播，从触发事件的节点到其父节点，最后直到文档根节点。最后，当某个特定节点上注册的所有事件处理器按其顺序全部执行完毕后，窗口对象的事件处理器才有机会响应事件。

事件处理器任何时候都可以调用事件对象的 `stopPropagation` 方法，阻止事件进一步传播。该方法有时很实用，例如，你将一个按钮放在另一个可点击元素中，但你不希望点击该按钮会激活外部元素的点击行为。

下面的示例代码将 `mousedown` 处理器注册到按钮和其外部的段落节点上。在按钮上点击鼠标右键，按钮的处理器会调用 `stopPropagation` ，调度段落上的事件处理器执行。当点击鼠标其他键时，两个处理器都会执行。

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  let para = document.querySelector("p");
  let button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Handler for paragraph.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Handler for button.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

大多数事件对象都有`target`属性，指的是事件来源节点。你可以根据该属性防止无意中处理了传播自其他节点的事件。

我们也可以使用`target`属性来创建出特定类型事件的处理网络。例如，如果一个节点中包含了很长的按钮列表，比较方便的处理方式是在外部节点上注册一个点击事件处理器，并根据事件的`target`属性来区分用户按下了哪个按钮，而不是为每个按钮都注册独立的事件处理器。

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", event => {
    if (event.target.nodeName == "BUTTON") {
      console.log("Clicked", event.target.textContent);
    }
  });
</script>
```

默认动作

大多数事件都有与其关联的默认动作。若点击链接，就会跳转到链接目标。若点击向下的箭头，浏览器会向下翻页。若右击鼠标，可以得到一个上下文菜单等。

对于大多数类型的事件，JavaScript 事件处理器会在默认行为发生之前调用。若事件处理器不希望执行默认行为（通常是因为已经处理了该事件），会调用`preventDefault`事件对象的方法。

你可以实现你自己的键盘快捷键或交互式菜单。你也可以干扰用户期望的行为。例如，这里实现一个无法跳转的链接。

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

除非你有非常充足的理由，否则不要这样做。当预期的行为被打破时，使用你的页面的人会感到不快。

在有些浏览器中，你完全无法拦截某些事件。比如在 Chrome 中，关闭键盘快捷键（`CTRL-W` 或 `COMMAND-W`）无法由 JavaScript 处理。

按键事件

当按下键盘上的按键时，浏览器会触发 `"keydown"` 事件。当松开按键时，会触发 `"keyup"` 事件。

```
<p>This page turns violet when you hold the V key.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", event => {
    if (event.key == "v") {
      document.body.style.background = "";
    }
  });
</script>
```

尽管从 `keydown` 这个事件名上看应该是物理按键按下时触发，但当持续按下某个按键时，会循环触发该事件。有时，你想谨慎对待它。例如，如果您在按下某个按键时向 DOM 添加按钮，并且在释放按键时再次将其删除，则可能会在按住某个按键的时间过长时，意外添加数百个按钮。

该示例查看了事件对象的 `key` 属性，来查看事件关于哪个键。 该属性包含一个字符串，对于大多数键，它对应于按下该键时将键入的内容。 对于像 `Enter` 这样的特殊键，它包含一个用于命名键的字符串（在本例中为 `"Enter"`）。 如果你按住一个键的同时按住 `Shift` 键，这也可能影响键的名称 - `"v"` 变为 `"V"`，`"1"` 可能变成 `"!"`，这是按下 `Shift-1` 键 在键盘上产生的东西。

诸如 `shift`、`ctrl`、`alt` 和 `meta`（Mac 上的 `command`）之类的修饰按键会像普通按键一样产生事件。但在查找组合键时，你也可以查看键盘和鼠标事件的 `shiftKey`、`ctrlKey`、`altKey` 和 `metaKey` 属性来判断这些键是否被按下。

```
<p>Press Ctrl-Space to continue.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == " " && event.ctrlKey) {
      console.log("Continuing!");
    }
  });
</script>
```

按键事件发生的 DOM 节点取决于按下按键时具有焦点的元素。 大多数节点不能拥有焦点，除非你给他们一个`tabindex`属性，但像链接，按钮和表单字段可以。 我们将在第 18 章中回顾表单字段。 当没有特别的焦点时，`document.body`充当按键事件的目标节点。

当用户键入文本时，使用按键事件来确定正在键入的内容是有问题的。 某些平台，尤其是 Android 手机上的虚拟键盘，不会触发按键事件。 但即使你有一个老式键盘，某些类型的文本输入也不能直接匹配按键，例如其脚本不适合键盘的人所使用的 IME（“输入法编辑器”）软件 ，其中组合多个热键来创建字符。

要注意什么时候输入了内容，每当用户更改其内容时，可以键入的元素（例如`<input>`和`<textarea>`标签）触发`"input"`事件。为了获得输入的实际内容，最好直接从焦点字段中读取它。 第 18 章将展示如何实现。

指针事件

目前有两种广泛使用的方式，用于指向屏幕上的东西：鼠标（包括类似鼠标的设备，如触摸板和轨迹球）和触摸屏。 它们产生不同类型的事件。

鼠标点击

点击鼠标按键会触发一系列事件。`"mousedown"`事件和`"mouseup"`事件类似于`"keydown"`和`"keyup"`事件，当鼠标按钮按下或释放时触发。当事件发生时，由鼠标指针下方的 DOM 节点触发事件。

在`mouseup`事件后，包含鼠标按下与释放的特定节点会触发`"click"`事件。例如，如果我在一个段落上按下鼠标，移动到另一个段落上释放鼠标，`"click"`事件会发生在包含这两个段落的元素上。

若两次点击事件触发时机接近，则在第二次点击事件之后，也会触发`"dbclick"`（双击，double-click）事件。

为了获得鼠标事件触发的精确信息，你可以查看事件中的`clientX`和`clientY`属性，包含了事件相对于窗口左上角的坐标（以像素为单位）。或`pageX`和`pageY`，它们相对于整个文档的左上角（当窗口被滚动时可能不同）。

下面的代码实现了简单的绘图程序。每次点击文档时，会在鼠标指针下添加一个点。还有一个稍微优化的绘图程序，请参见第 19 章。

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* rounds corners */
    background: blue;
    position: absolute;
  }
</style>
<script>
  window.addEventListener("click", event => {
    let dot = document.createElement("div");
    dot.className = "dot";
    dot.style.left = (event.pageX - 4) + "px";
    dot.style.top = (event.pageY - 4) + "px";
    document.body.appendChild(dot);
  });
</script>
```

鼠标移动

每次鼠标移动时都会触发 `"mousemove"` 事件。该事件可用于跟踪鼠标位置。当实现某些形式的鼠标拖拽功能时，该事件非常有用。

举一个例子，下面的程序展示一条栏，并设置一个事件处理器，当向左拖动这个栏时，会使其变窄，若向右拖动则变宽。

```
<p>Drag the bar to change its width:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
  let lastX; // Tracks the last observed mouse X position
  let bar = document.querySelector("div");
  bar.addEventListener("mousedown", event => {
    if (event.button == 0) {
      lastX = event.clientX;
      window.addEventListener("mousemove", moved);
      event.preventDefault(); // Prevent selection
    }
  });


  function moved(event) {
    if (event.buttons == 0) {
      window.removeEventListener("mousemove", moved);
    } else {
      let dist = event.clientX - lastX;
      let newWidth = Math.max(10, bar.offsetWidth + dist);
      bar.style.width = newWidth + "px";
      lastX = event.clientX;
    }
  }
</script>
```

请注意，`mousemove` 处理器注册在窗口对象上。即使鼠标在改变窗口尺寸时在栏外侧移动，只要按住按钮，我们仍然想要更新其大小。

释放鼠标按键时，我们必须停止调整栏的大小。 为此，我们可以使用`buttons`属性（注意复数形式），它告诉我们当前按下的按键。 当它为零时，没有按下按键。 当按键被按住时，其值是这些按键的代码总和 - 左键代码为 1，右键为 2，中键为 4。 这样，您可以通过获取`buttons`的剩余值及其代码，来检查是否按下了给定按键。

请注意，这些代码的顺序与`button`使用的顺序不同，中键位于右键之前。 如前所述，一致性并不是浏览器编程接口的强项。

触摸事件

我们使用的图形浏览器的风格，是考虑到鼠标界面的情况下而设计的，那个时候触摸屏非常罕见。 为了使网络在早期的触摸屏手机上“工作”，在某种程度上，这些设备的浏览器假装触摸事件是鼠标事件。 如果你点击你的屏幕，你会得到`'mousedown'`，`'mouseup'`和`'click'`事件。

但是这种错觉不是很健壮。 触摸屏与鼠标的工作方式不同：它没有多个按钮，当手指不在屏幕上时不能跟踪手指（来模拟`"mousemove"`），并且允许多个手指同时在屏幕上。

鼠标事件只涵盖了简单情况下的触摸交互 - 如果您为按钮添加`"click"`处理器，触摸用户仍然可以使用它。 但是像上一个示例中的可调整大小的栏在触摸屏上不起作用。

触摸交互触发了特定的事件类型。 当手指开始触摸屏幕时，您会看到`'touchstart'`事件。 当它在触摸中移动时，触发`"touchmove"`事件。 最后，当它停止触摸屏幕时，您会看到`"touchend"`事件。

由于许多触摸屏可以同时检测多个手指，这些事件没有与其关联的一组坐标。 相反，它们的事件对象拥有`touches`属性，它拥有一个类数组对象，每个对象都有自己的`clientX`，`clientY`，`pageX`和`pageY`属性。

你可以这样，在每个触摸手指周围显示红色圆圈。

```
<style>
  dot { position: absolute; display: block;
        border: 2px solid red; border-radius: 50px;
        height: 100px; width: 100px; }
</style>
<p>Touch this page</p>
<script>
  function update(event) {
    for (let dot; dot = document.querySelector("dot");) {
      dot.remove();
    }
    for (let i = 0; i < event.touches.length; i++) {
      let {pageX, pageY} = event.touches[i];
      let dot = document.createElement("dot");
      dot.style.left = (pageX - 50) + "px";
      dot.style.top = (pageY - 50) + "px";
      document.body.appendChild(dot);
    }
  }
  window.addEventListener("touchstart", update);
  window.addEventListener("touchmove", update);
  window.addEventListener("touchend", update);
</script>
```

您经常希望在触摸事件处理器中调用`preventDefault`，来覆盖浏览器的默认行为（可能包括在滑动时滚动页面），并防止触发鼠标事件，您也可能拥有它的处理器。

滚动事件

每当元素滚动时，会触发`scroll`事件。该事件用处极多，比如知道用户当前查看的元素（禁用用户视线以外的动画，或向邪恶的指挥部发送监视报告），或展示一些滚动的迹象（通过高亮表格的部分内容，或显示页码）。

以下示例在文档上方绘制一个进度条，并在您向下滚动时更新它来填充：

```
<style>
  #progress {
    border-bottom: 2px solid blue;
    width: 0;
    position: fixed;
    top: 0; left: 0;
  }
</style>
<div id="progress"></div>
<script>
  // Create some content
  document.body.appendChild(document.createTextNode(
    "supercalifragilisticexpialidocious ".repeat(1000)));

  let bar = document.querySelector("#progress");
  window.addEventListener("scroll", () => {
    let max = document.body.scrollHeight - innerHeight;
    bar.style.width = `${(pageYOffset / max) * 100}%`;
  });
</script>
```

将元素的 `position` 属性指定为 `fixed` 时，其行为和 `absolute` 很像，但可以防止在文档滚动时期跟着文档一起滚动。其效果是让我们的进度条呆在最顶上。 改变其宽度来指示当前进度。 在设置宽度时，我们使用 `%` 而不是 `px` 作为单位，使元素的大小相对于页面宽度。

`innerHeight` 全局绑定是窗口高度，我们必须要减去滚动条的高度。你点击文档底部的时候是无法继续滚动的。对于窗口高度来说，也存在 `innerWidth`。使用 `pageYOffset`（当前滚动位置）除以最大滚动位置，并乘以 100，就可以得到进度条长度。

调用滚动事件的 `preventDefault` 无法阻止滚动。实际上，事件处理器是在进行滚动之后才触发的。

焦点事件

当元素获得焦点时，浏览器会触发其上的 `focus` 事件。当失去焦点时，元素会获得 `blur` 事件。

与前文讨论的事件不同，这两个事件不会传播。子元素获得或失去焦点时，不会激活父元素的处理器。

下面的示例中，文本域在拥有焦点时会显示帮助文本。

```
<p>Name: <input type="text" data-help="Your full name"></p>
<p>Age: <input type="text" data-help="Your age in years"></p>
<p id="help"></p>

<script>
  let help = document.querySelector("#help");
  let fields = document.querySelectorAll("input");
  for (let field of Array.from(fields)) {
    field.addEventListener("focus", event => {
      let text = event.target.getAttribute("data-help");
      help.textContent = text;
    });
    field.addEventListener("blur", event => {
      help.textContent = "";
    });
  }
</script>
```

当用户从浏览器标签或窗口移开时，窗口对象会收到 `focus` 事件，当移动到标签或窗口上时，则收到 `blur` 事件。

加载事件

当界面结束装载时，会触发窗口对象和文档 `body` 对象的 `"load"` 事件。该事件通常用于在当整个文档构建完成时，进行初始化。请记住 `<script>`标签的内容是一遇到就执行的。这可能太早了，比如有时脚本需要处理在 `<script>` 标签后出现的内容。

诸如`image`或`script`这类会装载外部文件的标签都有`load`事件，指示其引用文件装载完毕。类似于焦点事件，装载事件是不会传播的。

当页面关闭或跳转（比如跳转到一个链接）时，会触发`beforeunload`事件。该事件用于防止用户突然关闭文档而丢失工作结果。你无法使用`preventDefault`方法阻止页面卸载。它通过从处理器返回非空值来完成。当你这样做时，浏览器会通过显示一个对话框，询问用户是否关闭页面的对话框中。该机制确保用户可以离开，即使在那些想要留住用户，强制用户看广告的恶意页面上，也是这样。

事件和事件循环

在事件循环的上下文中，如第 11 章中所述，浏览器事件处理器的行为，类似于其他异步通知。 它们是在事件发生时调度的，但在它们有机会运行之前，必须等待其他正在运行的脚本完成。

仅当没有别的事情正在运行时，才能处理事件，这个事实意味着，如果事件循环与其他工作捆绑在一起，任何页面交互（通过事件发生）都将延迟，直到有时间处理它为止。 因此，如果您安排了太多工作，无论是长时间运行的事件处理器还是大量短时间运行的工作，该页面都会变得缓慢且麻烦。

如果您想在背后做一些耗时的事情而不会冻结页面，浏览器会提供一些名为 Web Worker 的东西。 Web Worker 是一个 JavaScript 过程，与主脚本一起在自己的时间线上运行。

想象一下，计算一个数字的平方运算是一个重量级的，长期运行的计算，我们希望在一个单独的线程中执行。 我们可以编写一个名为`code/squareworker.js`的文件，通过计算平方并发回消息来响应消息：

```
addEventListener("message", event => {
  postMessage(event.data * event.data);
});
```

为了避免多线程触及相同数据的问题，Web Worker 不会将其全局作用域或任何其他数据与主脚本的环境共享。 相反，你必须通过来回发送消息与他们沟通。

此代码会生成一个运行该脚本的 Web Worker，向其发送几条消息并输出响应。

```
let squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", event => {
  console.log("The worker responded:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

函数 `postMessage` 会发送一条消息，触发接收方的 `message` 事件。创建工作单元的脚本通过 `Worker` 对象收发消息，而 `worker` 则直接向其全局作用域发送消息，或监听其消息。只有可以表示为 JSON 的值可以作为消息发送 - 另一方将接收它们的副本，而不是值本身。

定时器

我们在第 11 章中看到了 `setTimeout` 函数。 它会在给定的毫秒数之后，调度另一个函数在稍后调用。

有时读者需要取消调度的函数。可以存储 `setTimeout` 的返回值，并将作为参数调用 `clearTimeout` 。

```
let bombTimer = setTimeout(() => {
  console.log("BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% chance
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

函数 `cancelAnimationFrame` 作用与 `clearTimeout` 相同，使用 `requestAnimationFrame` 的返回值调用该函数，可以取消帧（假定函数还没有被调用）。

还有 `setInterval` 和 `clearInterval` 这种相似的函数，用于设置计时器，每隔一定毫秒数重复执行一次。

```
let ticks = 0;
let clock = setInterval(() => {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

降频

某些类型的事件可能会连续、迅速触发多次（例如 `mousemove` 和 `scroll` 事件）。处理这类事件时，你必须小心谨慎，防止处理任务耗时过长，否则处理器会占据过多事件，导致用户与文档交互变得非常慢。

若你需要在这类处理器中编写一些重要任务，可以使用 `setTimeout` 来确保不会频繁进行这些任务。我们通常称之为“事件降频（Debounce）”。有许多方法可以完成该任务。

在第一个示例中，当用户输入某些字符时，我们想要有所反应，但我们不想在每个按键事件中立即处理该任务。当用户输入过快时，我们希望暂停一下然后进行处理。我们不是立即在事件处理器中执行动作，而是设置一个定时器。我们也会清除上一次的定时器（如果有），因此当两个事件触发间隔过短（比定时器延时短），就会取消上一次事件设置的定时器。

```
<textarea>Type something here...</textarea>
<script>
  let textarea = document.querySelector("textarea");
  let timeout;
  textarea.addEventListener("input", () => {
    clearTimeout(timeout);
    timeout = setTimeout(() => console.log("Typed!"), 500);
  });
</script>
```

将`undefined`传递给`clearTimeout`或在一个已结束的定时器上调用`clearTimeout`是没有效果的。因此，我们不需要关心何时调用该方法，只需要每个事件中都这样做即可。

如果我们想要保证每次响应之间至少间隔一段时间，但不希望每次事件发生时都重置定时器，而是在一连串事件连续发生时能够定时触发响应，那么我们可以使用一个略有区别的方法来解决问题。例如，我们想要响应`"mousemove"`事件来显示当前鼠标坐标，但频率只有 250ms。

```
<script>
  let scheduled = null;
  window.addEventListener("mousemove", event => {
    if (!scheduled) {
      setTimeout(() => {
        document.body.textContent =
          `Mouse at ${scheduled.pageX}, ${scheduled.pageY}`;
        scheduled = null;
      }, 250);
    }
    scheduled = event;
  });
</script>
```

本章小结

事件处理器可以检测并响应发生在我们的 Web 页面上的事件。`addEventListener`方法用于注册处理器。

每个事件都有标识事件的类型（`keydown`、`focus`等）。大多数方法都会在特定 DOM 元素上调用，接着向其父节点传播，允许每个父元素的处理器都能处理这些事件。

JavaScript 调用事件处理器时，会传递一个包含事件额外信息的事件对象。该对象也有方法支持停止进一步传播（`stopPropagation`），也支持阻止浏览器执行事件的默认处理器（`preventDefault`）。

按下键盘按键时会触发`keydown`和`keyup`事件。按下鼠标按钮时，会触发`mousedown`、`mouseup`和`click`事件。移动鼠标会触发`mousemove`事件。触摸屏交互会导致`"touchstart"`，`"touchmove"`和`"touchend"`事件。

我们可以通过`scroll`事件监测滚动行为，可以通过`focus`和`blur`事件监控焦点改变。当文档完成加载后，会触发窗口的`load`事件。

习题

气球

编写一个显示气球的页面（使用气球 emoji，`\ud83c\udf88`）。 当你按下上箭头时，它应该变大（膨胀）10%，而当你按下下箭头时，它应该缩小（放气）10%。

您可以通过在其父元素上设置`font-size` CSS 属性（`style.fontSize`）来控制文本大小（emoji 是文本）。 请记住在该值中包含一个单位，例如像素（`10px`）。

箭头键的键名是`"ArrowUp"`和`"ArrowDown"`。确保按键只更改气球，而不滚动页面。

实现了之后，添加一个功能，如果你将气球吹过一定的尺寸，它就会爆炸。 在这种情况下，爆炸意味着将其替换为“爆炸 emoji，`\ud83d\udca5`”，并且移除事件处理器（以便您不能使爆炸变大变小）。

```
<p>&#x1f4a5;</p>
<script>
  // Your code here
</script>
```

鼠标轨迹

在 JavaScript 早期，有许多主页都会在页面上使用大量的动画，人们想出了许多该语言的创造性用法。

其中一种是“鼠标踪迹”，也就是一系列的元素，随着你在页面上移动鼠标，它会跟着你的鼠标指针。

在本习题中实现鼠标轨迹的功能。使用绝对定位、固定尺寸的`<div>`元素，背景为黑色（请参考鼠标点击一节中的示例）。创建一系列此类元素，当鼠标移动时，伴随鼠标指针显示它们。

有许多方案可以实现我们所需的功能。你可以根据你的需要实现简单的或复杂的方法。简单的解决方案是保存固定鼠标的轨迹元素并循环使用它们，每次`mousemove`事件触发时将下一个元素移动到鼠标当前位置。

```
<style>
  .trail { /* className for the trail elements */
    position: absolute;
    height: 6px; width: 6px;
    border-radius: 3px;
    background: teal;
  }
  body {
    height: 300px;
  }
</style>

<script>
  // Your code here.
</script>
```

选项卡

选项卡面板广泛用于用户界面。它支持用户通过选择元素上方的很多突出的选项卡来选择一个面板。

本习题中，你必须实现一个简单的选项卡界面。编写 `asTabs` 函数，接受一个 DOM 节点并创建选项卡界面来展现该节点的子元素。该函数应该在顶层节点中插入大量 `<button>` 元素，与每个子元素一一对应，按钮文本从子节点的 `data-tabname` 中获取。除了显示一个初始子节点，其他子节点都应该隐藏（将 `display` 样式设置成 `none`），并通过点击按钮来选择当前显示的节点。

当它生效时将其扩展，为当前选中的选项卡，将按钮的样式设为不同的，以便明确选择了哪个选项卡。

```
<tab-panel>
  <div data-tabname="one">Tab one</div>
  <div data-tabname="two">Tab two</div>
  <div data-tabname="three">Tab three</div>
</tab-panel>
<script>
  function asTabs(node) {
    // Your code here.
  }
  asTabs(document.querySelector("tab-panel"));
</script>
```

十六、项目：平台游戏

> 原文：Project: A Platform Game
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 所有现实都是游戏。
>
> Iain Banks，《The Player of Games》



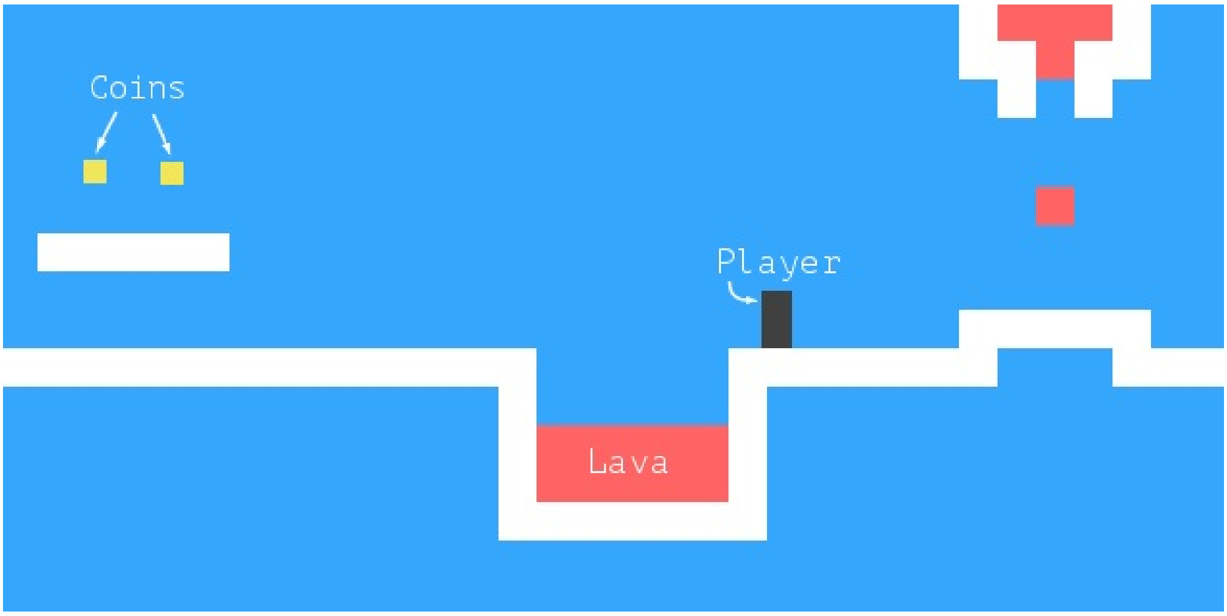
我最初对电脑的痴迷，就像许多小孩一样，与电脑游戏有关。我沉迷在那个计算机所模拟出的小小世界中，我可以操纵这个世界，我同时也沉迷在那些尚未展开的故事之中。但我沉迷其中并不是因为游戏实际描述的故事，而是因为我可以充分发挥我的想象力，去构思故事的发展。

我并不希望任何人把编写游戏作为自己的事业。就像音乐产业中，那些希望加入这个行业的热忱年轻人与实际的人才需求之间存在巨大的鸿沟，也因此产生了一个极不健康的就业环境。不过，把编写游戏作为乐趣还是相当不错的。

本章将会介绍如何实现一个小型平台游戏。平台游戏（或者叫作“跳爬”游戏）要求玩家操纵一个角色在世界中移动，这种游戏往往是二维的，而且采用单一侧面作为观察视角，玩家可以来回跳跃。

游戏

我们游戏大致基于由 Thomas Palef 开发的 Dark Blue。我之所以选择了这个游戏，是因为这个游戏既有趣又简单，而且不需要编写大量代码。该游戏看起来如下页图所示。



黑色的方块表示玩家，玩家任务是收集黄色的方块（硬币），同时避免碰到红色素材（“岩浆”）。当玩家收集完所有硬币后就可以过关。

玩家可以使用左右方向键移动，并使用上方向键跳跃。跳跃正是这个游戏角色的特长。玩家可以跳跃到数倍于自己身高的地方，也可以在半空中改变方向。虽然这样不切实际，但这有助于玩家感觉自己在直接控制屏幕上那个自己的化身。

该游戏包含一个固定的背景，使用网格方式进行布局，可可移动元素则覆盖在背景之上。网格中的元素可能是空气、固体或岩浆。可可移动元素是玩家、硬币或者某一块岩浆。这些元素的位置不限于网格，它们的坐标可以是分数，允许平滑运动。

实现技术

我们会使用浏览器的 DOM 来展示游戏界面，我们会通过处理按键事件来读取用户输入。

与屏幕和键盘相关的代码只是实现游戏代码中的很小一部分。由于所有元素都只是彩色方块，因此绘制方法并不复杂。我们为每个元素创建对应的 DOM 元素，并使用样式来为其指定背景颜色、尺寸和位置。

由于背景是由不会改变的方块组成的网格，因此我们可以使用表格来展示背景。自由可移动元素可以使用绝对定位元素来覆盖。

游戏和某些程序应该在不产生明显延迟的情况下绘制动画并响应用户输入，性能是非常重要的。尽管 DOM 最初并非为高性能绘图而设计，但实际上 DOM 的性能表现得比我们想象中要好得多。读者已经在第 13 章中看过一些动画，在现代机器中，即使我们不怎么考虑性能优化，像这种简单的游戏也可以流畅运行。

在下一章中，我们会研究另一种浏览器技术 —— `<canvas>` 标签。该标签提供了一种更为传统的图像绘制方式，直接处理形状和像素而非 DOM 元素。

关卡

我们需要一种人类可读的、可编辑的方法来指定关卡。因为一切最开始都可以在网格，所以我们可以使用大型字符串，其中每个字符代表一个元素，要么是背景网格的一部分，要么是可移动元素。

小型关卡的平面图可能是这样的：

```
var simpleLevelPlan = `
......................
..#................#..
..#..............=.#..
..#.........o.o....#..
..#.@......#####...#..
..#####............#..
......#++++++++++++#..
......##############..
......................`;
```

句号是空的位置，井号（`#`）字符是墙，加号是岩浆。玩家的起始位置是 AT 符号（`@`）。每个`o`字符都是一枚硬币，等号（`=`）是一块来回水平移动的岩浆块。

我们支持两种额外的可移动岩浆：管道符号（`|`）表示垂直移动的岩浆块，而`v`表示下落的岩浆块 —— 这种岩浆块也是垂直移动，但不会来回弹跳，只会向下移动，直到遇到地面才会直接回到其起始位置。

整个游戏包含了许多关卡，玩家必须完成所有关卡。每关的过关条件是玩家需要收集所有硬币。如果玩家碰到岩浆，当前关卡会恢复初始状态，而玩家可以再次尝试过关。

读取关卡

下面的类存储了关卡对象。它的参数应该是定义关卡的字符串。

```
class Level {
  constructor(plan) {
    let rows = plan.trim().split("\n").map(l => [...l]);
    this.height = rows.length;
    this.width = rows[0].length;
    this.startActors = [];
    this.rows = rows.map((row, y) => {
      return row.map((ch, x) => {
        let type = levelChars[ch];
        if (typeof type == "string") return type;
        this.startActors.push(
          type.create(new Vec(x, y), ch));
        return "empty";
      });
    });
  }
}
```

`trim` 方法用于移除平面图字符串起始和终止处的空白。这允许我们的示例平面图以换行开始，以便所有行都在彼此的正下方。其余的字符串由换行符拆分，每一行扩展到一个数组中，生成了字符数组。

因此，`rows` 包含字符数组、平面图的行。我们可以从中得出水平宽度和高度。但是我们仍然必须将可移动元素与背景网格分开。我们将其称为角色（Actor）。它们将存储在一个对象数组中。背景将是字符串的数组的数组，持有字段类型，如 `"empty"`，`"wall"`，或 `"lava"`。

为了创建这些数组，我们在行上映射，然后在它们的内容上进行映射。请记住，`map` 将数组索引作为第二个参数传递给映射函数，它告诉我们给定字符的 `x` 和 `y` 坐标。游戏中的位置将存储为一对坐标，左上角为 `0, 0`，并且每个背景方块为 1 单位高和宽。

为了解释平面图中的字符，`Level` 构造器使用 `levelChars` 对象，它将背景元素映射为字符串，角色字符映射为类。当 `type` 是一个角色类时，它的 `create` 静态方法用于创建一个对象，该对象被添加到 `startActors`，映射函数为这个背景方块返回 `"empty"`。

角色的位置存储为一个 `Vec` 对象，它是二维向量，一个具有 `x` 和 `y` 属性的对象，像第六章一样。

当游戏运行时，角色将停在不同的地方，甚至完全消失（就像硬币被收集时）。我们将使用一个 `State` 类来跟踪正在运行的游戏的状态。

```
class State {
  constructor(level, actors, status) {
    this.level = level;
    this.actors = actors;
    this.status = status;
  }

  static start(level) {
    return new State(level, level.startActors, "playing");
  }

  get player() {
    return this.actors.find(a => a.type == "player");
  }
}
```

当游戏结束时，`status`属性将切换为`"lost"`或`"won"`。

这又是一个持久性数据结构，更新游戏状态会创建新状态，并使旧状态保持完整。

角色

角色对象表示，游戏中给定可移动元素的当前位置和状态。所有的角色对象都遵循相同的接口。它们的`pos`属性保存元素的左上角坐标，它们的`size`属性保存其大小。

然后，他们有`update`方法，用于计算给定时间步长之后，他们的新状态和位置。它模拟了角色所做的事情：响应箭头键并且移动，因岩浆而来回弹跳，并返回新的更新后的角色对象。

`type`属性包含一个字符串，该字符串指定了角色类型：`"player"`，`"coin"`或者`"lava"`。这在绘制游戏时是有用的，为角色绘制的矩形的外观基于其类型。

角色类有一个静态的`create`方法，它由`Level`构造器使用，用于从关卡平面图中的字符中，创建一个角色。它接受字符本身及其坐标，这是必需的，因为`Lava`类处理几个不同的字符。

这是我们将用于二维值的`Vec`类，例如角色的位置和大小。

```
class Vec {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  plus(other) {
    return new Vec(this.x + other.x, this.y + other.y);
  }
  times(factor) {
    return new Vec(this.x * factor, this.y * factor);
  }
}
```

`times`方法用给定的数字来缩放向量。当我们需要将速度向量乘时间间隔，来获得那个时间的行走距离时，这就有用了。

不同类型的角色拥有他们自己的类，因为他们的行为非常不同。让我们定义这些类。稍后我们将看看他们的`update`方法。

玩家类拥有`speed`属性，存储了当前速度，来模拟动量和重力。

```
class Player {
  constructor(pos, speed) {
    this.pos = pos;
    this.speed = speed;
  }

  get type() { return "player"; }

  static create(pos) {
    return new Player(pos.plus(new Vec(0, -0.5)),
                      new Vec(0, 0));
  }
}

Player.prototype.size = new Vec(0.8, 1.5);
```

因为玩家高度是一个半格子，因此其初始位置相比于`@`字符出现的位置要高出半个格子。这样一来，玩家角色的底部就可以和其出现的方格底部对齐。

`size`属性对于`Player`的所有实例都是相同的，因此我们将其存储在原型上，而不是实例本身。我们可以使用一个类似`type`的读取器，但是每次读取属性时，都会创建并返回一个新的`Vec`对象，这将是浪费的。（字符串是不可变的，不必在每次求值时重新创建。）

构造`Lava`角色时，我们需要根据它所基于的字符来初始化对象。动态岩浆以其当前速度移动，直到它碰到障碍物。这个时候，如果它拥有`reset`属性，它会跳回到它的起始位置（滴落）。如果没有，它会反转它的速度并以另一个方向继续（弹跳）。

`create`方法查看`Level`构造器传递的字符，并创建适当的岩浆角色。

```
class Lava {
  constructor(pos, speed, reset) {
    this.pos = pos;
    this.speed = speed;
    this.reset = reset;
  }

  get type() { return "lava"; }

  static create(pos, ch) {
    if (ch == "=") {
      return new Lava(pos, new Vec(2, 0));
    } else if (ch == "|") {
      return new Lava(pos, new Vec(0, 2));
    } else if (ch == "v") {
      return new Lava(pos, new Vec(0, 3), pos);
    }
  }
}

Lava.prototype.size = new Vec(1, 1);
```

`Coin`对象相对简单，大多时候只需要待在原地即可。但为了使游戏更加有趣，我们让硬币轻微摇晃，也就是会在垂直方向上小幅度来回移动。每个硬币对象都存储了其基本位置，同时使用`wobble`属性跟踪图像跳动幅度。这两个属性同时决定了硬币的实际位置（存储在`pos`属性中）。

```
class Coin {
  constructor(pos, basePos, wobble) {
    this.pos = pos;
    this.basePos = basePos;
    this.wobble = wobble;
  }

  get type() { return "coin"; }

  static create(pos) {
    let basePos = pos.plus(new Vec(0.2, 0.1));
    return new Coin(basePos, basePos,
                    Math.random() * Math.PI * 2);
  }
}

Coin.prototype.size = new Vec(0.6, 0.6);
```

第十四章中，我们知道了`Math.sin`可以计算出圆的`y`坐标。因为我们沿着圆移动，因此`y`坐标会以平滑的波浪形式来回移动，正弦函数在实现波浪形移动中非常实用。

为了避免出现所有硬币同时上下移动，每个硬币的初始阶段都是随机的。由`Math.sin`产生的波长是`2π`。我们可以将`Math.random`的返回值乘以`2π`，计算出硬币波形轨迹的初始位置。

现在我们可以定义`levelChars`对象，它将平面图字符映射为背景网格类型，或角色类。

```
const levelChars = {
  ".": "empty", "#": "wall", "+": "lava",
  "@": Player, "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};
```

这给了我们创建`Level`实例所需的所有部件。

```
let simpleLevel = new Level(simpleLevelPlan);
console.log(`${simpleLevel.width} by ${simpleLevel.height}`);
// → 22 by 9
```

上面一段代码的任务是将特定关卡显示在屏幕上，并构建关卡中的时间与动作。

成为负担的封装

本章中大多数代码并没有过多考虑封装。首先，封装需要耗费额外精力。封装使得程序变得更加庞大，而且会引入额外的概念和接口。我尽量将程序的体积控制在较小的范围之内，避免读者因为代码过于庞大而走神。

其次，游戏中的大量元素是紧密耦合在一起的，如果其中一个元素行为改变，其他的元素很有可能也会发生变化。我们需要根据游戏的工作细节来为元素之间设计大量接口。这使得接口的效果不是很好。每当你改变系统中的某一部分时，由于其他部分的接口可能没有考虑到新的情况，因此你需要关心这一修改是否会影响到其他部分的代码。

系统中的某些分割点可以通过严格的接口对系统进行合理的划分，但某些分割点则不是如此。尝试去封装某些本没有合理边界的代码必然会导致浪费大量精力。当你犯下这种大错之际，你就会注意到你的接口变得庞大臃肿，而且随着程序不断演化，你需要频繁修改这些接口。

我们会封装的一部分代码是绘图子系统。其原因是我们会在下一章中使用另一种方式来展示相同的游戏。通过将绘图代码隐藏在接口之后，我们可以在下一章中使用相同的游戏程序，只需要插入新的显示模块即可。

绘图

我们通过定义一个“显示器”对象来封装绘图代码，该对象显示指定关卡，以及状态。本章定义的显示器类型名为`DOMDisplay`，因为该类型使用简单的 DOM 元素来显示关卡。

我们会使用样式表来设定实际的颜色以及其他构建游戏中所需的固定的属性。创建这些属性时，我们可以直接对元素的`style`属性进行赋值，但这会使得游戏代码变得冗长。

下面的帮助函数提供了一种简洁的方法，来创建元素并赋予它一些属性和子节点：

```
function elt(name, attrs, ...children) {
  let dom = document.createElement(name);
  for (let attr of Object.keys(attrs)) {
    dom.setAttribute(attr, attrs[attr]);
  }
  for (let child of children) {
    dom.appendChild(child);
  }
  return dom;
}
```

我们创建显示器对象时需要指定其父元素，显示器将会创建在该父元素上，同时还需指定一个关卡对象。

```
class DOMDisplay {
  constructor(parent, level) {
    this.dom = elt("div", {class: "game"}, drawGrid(level));
    this.actorLayer = null;
    parent.appendChild(this.dom);
  }

  clear() { this.dom.remove(); }
}
```

由于关卡的背景网格不会改变，因此只需要绘制一次即可。角色则需要在每次刷新显示时进行重绘。`drawFame`需要使用`actorLayer`属性来跟踪已保存角色的动作，因此我们可以轻松移除或替换这些角色。

我们的坐标和尺寸以网格单元为单位跟踪，也就是说尺寸或距离中的 1 单元表示一个单元格。在设置像素级尺寸时，我们需要将坐标按比例放大，如果游戏中的所有元素只占据一个方格中的一个像素，那将是多么可笑。而 `scale` 绑定会给出一个单元格在屏幕上实际占据的像素数目。

```
const scale = 20;

function drawGrid(level) {
  return elt("table", {
    class: "background",
    style: `width: ${level.width * scale}px`
  }, ...level.rows.map(row =>
    elt("tr", {style: `height: ${scale}px`},
        ...row.map(type => elt("td", {class: type})))
  ));
}
```

前文提及过，我们使用 `<table>` 元素来绘制背景。这非常符合关卡中 `grid` 属性的结构。网格中的每一行对应表格中的一行（ `<tr>` 元素）。网格中的每个字符串对应表格单元格（ `<td>` ）元素的类型名。扩展（三点）运算符用于将子节点数组作为单独的参数传给 `elt` 。

下面的 CSS 使表格看起来像我们想要的背景：

```
.background    { background: rgb(52, 166, 251);
                 table-layout: fixed;
                 border-spacing: 0;              }
.background td { padding: 0;                     }
.lava          { background: rgb(255, 100, 100); }
.wall          { background: white;              }
```

其中某些属性（border-spacing和padding）用于取消一些我们不想保留的表格默认行为。我们不希望在单元格之间或单元格内部填充多余的空白。

其中 `background` 规则用于设置背景颜色。CSS中可以使用两种方式来指定颜色，一种方法是使用单词（ `white` ），另一种方法是使用形如 `rgb(R,G,B)` 的格式，其中 `R` 表示颜色中的红色成分， `G` 表示绿色成分， `B` 表示蓝色成分，每个数字范围均为 0 到 255。因此在 `rgb(52,166,251)` 中，红色成分为 52，绿色为 166，而蓝色是 251。由于蓝色成分数值最大，因此最后的颜色会偏向蓝色。而你可以看到 `.lava` 规则中，第一个数字（红色）是最大的。

我们绘制每个角色时需要创建其对应的 DOM 元素，并根据角色属性来设置元素坐标与尺寸。这些值都需要与 `scale` 相乘，以将游戏中的尺寸单位转换为像素。

```
function drawActors(actors) {
  return elt("div", {}, ...actors.map(actor => {
    let rect = elt("div", {class: `actor ${actor.type}`});
    rect.style.width = `${actor.size.x * scale}px`;
    rect.style.height = `${actor.size.y * scale}px`;
    rect.style.left = `${actor.pos.x * scale}px`;
    rect.style.top = `${actor.pos.y * scale}px`;
    return rect;
  }));
}
```

为了赋予一个元素多个类别，我们使用空格来分隔类名。在下面展示的 CSS 代码中，`actor`类会赋予角色一个绝对坐标。我们将角色的类型名称作为额外的 CSS 类来设置这些元素的颜色。我们并没有再次定义`lava`类，因为我们可以直接复用前文为岩浆单元格定义的规则。

```
.actor  { position: absolute;            }
.coin   { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64);   }
```

`setState`方法用于使显示器显示给定的状态。它首先删除旧角色的图形，如果有的话，然后在他们的新位置上重新绘制角色。试图将 DOM 元素重用于角色，可能很吸引人，但是为了使它有效，我们需要大量的附加记录，来关联角色和 DOM 元素，并确保在角色消失时删除元素。因为游戏中通常只有少数角色，重新绘制它们开销并不大。

```
DOMDisplay.prototype.setState = function(state) {
  if (this.actorLayer) this.actorLayer.remove();
  this.actorLayer = drawActors(state.actors);
  this.dom.appendChild(this.actorLayer);
  this.dom.className = `game ${state.status}`;
  this.scrollPlayerIntoView(state);
};
```

我们可以将关卡的当前状态作为类名添加到包装器中，这样可以根据游戏胜负与否来改变玩家角色的样式。我们只需要添加 CSS 规则，指定祖先节点包含特定类的`player`元素的样式即可。

```
.lost .player {
  background: rgb(160, 64, 64);
}
.won .player {
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;
}
```

在遇到岩浆之后，玩家的颜色应该变成深红色，暗示着角色被烧焦了。当玩家收集完最后一枚硬币时，我们添加两个模糊的白色阴影来创建白色的光环效果，其中一个在左上角，一个在右上角。

我们无法假定关卡总是符合视口尺寸，它是我们在其中绘制游戏的元素。所以我们需要调用`scrollPlayerIntoView`来确保如果关卡在视口范围之外，我们可以滚动视口，确保玩家靠近视口的中央位置。下面的 CSS 样式为包装器的DOM元素设置了一个最大尺寸，以确保任何超出视口的元素都是不可见的。我们可以将外部元素的`position`设置为`relative`，因此该元素中的角色总是相对于关卡的左上角进行定位。

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

在`scrollPlayerIntoView`方法中，我们找出玩家的位置并更新其包装器元素的滚动坐标。我们可以通过操作元素的`scrollLeft`和`scrollTop`属性，当玩家接近视口边界时修改滚动坐标。

```
DOMDisplay.prototype.scrollPlayerIntoView = function(state) {
  let width = this.dom.clientWidth;
  let height = this.dom.clientHeight;
  let margin = width / 3;

  // The viewport
  let left = this.dom.scrollLeft, right = left + width;
  let top = this.dom.scrollTop, bottom = top + height;

  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5))
                         .times(scale);

  if (center.x < left + margin) {
    this.dom.scrollLeft = center.x - margin;
  } else if (center.x > right - margin) {
    this.dom.scrollLeft = center.x + margin - width;
  }
  if (center.y < top + margin) {
    this.dom.scrollTop = center.y - margin;
  } else if (center.y > bottom - margin) {
    this.dom.scrollTop = center.y + margin - height;
  }
};
```

找出玩家中心位置的代码展示了，我们如何使用`Vec`类型来写出相对可读的计算代码。为了找出玩家的中心位置，我们需要将左上角位置坐标加上其尺寸的一半。计算结果就是关卡坐标的中心位置。但是我们需要将结果向量乘以显示比例，以将坐标转换成像素级坐标。

接下来，我们对玩家的坐标进行一系列检测，确保其位置不会超出合法范围。这里需要注意的是这段代码有时候依然会设置无意义的滚动坐标，比如小于 0 的值或超出元素滚动区域的值。这是没问题的。DOM 会将其修改为可接受的值。如果我们将`scrollLeft`设置为`-10`，DOM 会将其修改为 0。

最简单的做法是每次重绘时都滚动视口，确保玩家总是在视口中央。但这种做法会导致画面剧烈晃动，当你跳跃时，视图会不断上下移动。比较合理的做法是在屏幕中央设置一个“中央区域”，玩家在这个区域内部移动时我们不会滚动视口。

我们现在能够显示小型关卡。

```
<link rel="stylesheet" href="css/game.css">

<script>
  let simpleLevel = new Level(simpleLevelPlan);
  let display = new DOMDisplay(document.body, simpleLevel);
  display.setState(State.start(simpleLevel));
</script>
```

我们可以在`link`标签中使用`rel="stylesheet"`，将一个 CSS 文件加载到页面中。文件`game.css`包含了我们的游戏所需的样式。

动作与冲突

现在我们是时候来添加一些动作了。这是游戏中最令人着迷的一部分。实现动作的最基本的方案（也是大多数游戏采用的）是将时间划分为一个个时间段，根据角色的每一步速度和时间长度，将元素移动一段距离。我们将以秒为单位测量时间，所以速度以单元每秒来表示。

移动东西非常简单。比较困难的一部分是处理元素之间的相互作用。当玩家撞到墙壁或者地板时，不可能简单地直接穿越过去。游戏必须注意特定的动作会导致两个对象产生碰撞，并需要采取相应措施。如果玩家遇到墙壁，则必须停下来，如果遇到硬币则必须将其收集起来。

想要解决通常情况下的碰撞问题是件艰巨任务。你可以找到一些我们称之为物理引擎的库，这些库会在二维或三维空间中模拟物理对象的相互作用。我们在本章中采用更合适的方案：只处理矩形物体之间的碰撞，并采用最简单的方案进行处理。

在移动角色或岩浆块时，我们需要测试元素是否会移动到墙里面。如果会的话，我们只要取消整个动作即可。而对动作的反应则取决于移动元素类型。如果是玩家则停下来，如果是岩浆块则反弹回去。

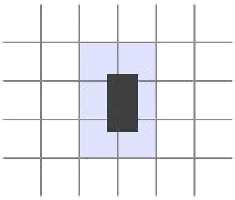
这种方法需要保证每一步之间的时间间隔足够短，确保能够在对象实际碰撞之前取消动作。如果时间间隔太大，玩家最后会悬浮在离地面很高的地方。另一种方法明显更好但更加复杂，即寻找到精确的碰撞点并将元素移动到那个位置。我们会采取最简单的方案，并确保减少动画之间的时间间隔，以掩盖其问题。

该方法用于判断某个矩形（通过位置与尺寸限定）是否会碰到给定类型的网格。

```
Level.prototype.touches = function(pos, size, type) {
  var xStart = Math.floor(pos.x);
  var xEnd = Math.ceil(pos.x + size.x);
  var yStart = Math.floor(pos.y);
  var yEnd = Math.ceil(pos.y + size.y);

  for (var y = yStart; y < yEnd; y++) {
    for (var x = xStart; x < xEnd; x++) {
      let isOutside = x < 0 || x >= this.width ||
                      y < 0 || y >= this.height;
      let here = isOutside ? "wall" : this.rows[y][x];
      if (here == type) return true;
    }
  }
  return false;
};
```

该方法通过对坐标使用 `Math.floor` 和 `Math.ceil`，来计算与身体重叠的网格方块集合。记住网格方块的大小是 `1x1` 个单位。通过将盒子的边上下颠倒，我们得到盒子接触的背景方块的范围。



我们通过查找坐标遍历网格方块，并在找到匹配的方块时返回 `true`。关卡之外的方块总是被当作 `"wall"`，来确保玩家不能离开这个世界，并且我们不会意外地尝试，在我们的“ `rows` 数组的边界之外读取。

状态的 `update` 方法使用 `touches` 来判断玩家是否接触岩浆。

```
State.prototype.update = function(time, keys) {
  let actors = this.actors
    .map(actor => actor.update(time, this, keys));
  let newState = new State(this.level, actors, this.status);
  if (newState.status != "playing") return newState;
  let player = newState.player;
  if (this.level.touches(player.pos, player.size, "lava")) {
    return new State(this.level, actors, "lost");
  }
  for (let actor of actors) {
    if (actor != player && overlap(actor, player)) {
      newState = actor.collide(newState);
    }
  }
  return newState;
};
```

它接受时间步长和一个数据结构，告诉它按下了哪些键。它所做的第一件事是调用所有角色的 `update` 方法，生成一组更新后的角色。角色也得到时间步长，按键，和状态，以便他们可以根据这些来更新。只有玩家才会读取按键，因为这是唯一由键盘控制的角色。

如果游戏已经结束，就不需要再做任何处理（游戏不能在输之后赢，反之亦然）。否则，该方法测试玩家是否接触背景岩浆。如果是这样的话，游戏就输了，我们就完了。最后，如果游戏实际上还在继续，它会查看其他玩家是否与玩家重叠。

`overlap` 函数检测角色之间的重叠。它需要两个角色对象，当它们触碰时返回 `true`，当它们沿 `X` 轴和 `Y` 轴重叠时，就是这种情况。

```
function overlap(actor1, actor2) {
  return actor1.pos.x + actor1.size.x > actor2.pos.x &&
         actor1.pos.x < actor2.pos.x + actor2.size.x &&
         actor1.pos.y + actor1.size.y > actor2.pos.y &&
         actor1.pos.y < actor2.pos.y + actor2.size.y;
}
```

如果任何角色重叠了，它的 `collide` 方法有机会更新状态。触碰岩浆角色将游戏状态设置为 `"lost"`，当你碰到硬币时，硬币就会消失，当这是最后一枚硬币时，状态就变成了 `"won"`。

```
Lava.prototype.collide = function(state) {
  return new State(state.level, state.actors, "lost");
};

Coin.prototype.collide = function(state) {
  let filtered = state.actors.filter(a => a != this);
  let status = state.status;
  if (!filtered.some(a => a.type == "coin")) status = "won";
  return new State(state.level, filtered, status);
};
```

角色的更新

角色对象的 `update` 方法接受时间步长、状态对象和 `keys` 对象作为参数。`Lava` 角色类型忽略 `keys` 对象。

```
Lava.prototype.update = function(time, state) {
  let newPos = this.pos.plus(this.speed.times(time));
  if (!state.level.touches(newPos, this.size, "wall")) {
    return new Lava(newPos, this.speed, this.reset);
  } else if (this.reset) {
    return new Lava(this.reset, this.speed, this.reset);
  } else {
    return new Lava(this.pos, this.speed.times(-1));
  }
};
```

它通过将时间步长乘上当前速度，并将其加到其旧位置，来计算新的位置。如果新的位置上没有障碍，它移动到那里。如果有障碍物，其行为取决于岩浆块的类型：滴落岩浆具有 `reset` 位置，当它碰到某物时，它会跳回去。跳跃岩浆将其速度乘以 `-1`，从而开始向相反的方向移动。

硬币使用它们的`act`方法来晃动。他们忽略了网格的碰撞，因为它们只是在它们自己的方块内部晃动。

```
const wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.update = function(time) {
  let wobble = this.wobble + time * wobbleSpeed;
  let wobblePos = Math.sin(wobble) * wobbleDist;
  return new Coin(this.basePos.plus(new Vec(0, wobblePos)),
                  this.basePos, wobble);
};
```

递增`wobble`属性来跟踪时间，然后用作`Math.sin`的参数，来找到波上的新位置。然后，根据其基本位置和基于波的偏移，计算硬币的当前位置。

还剩下玩家本身。玩家的运动对于每和轴单独处理，因为碰到地板不应阻止水平运动，碰到墙壁不应停止下降或跳跃运动。

```
const playerXSpeed = 7;
const gravity = 30;
const jumpSpeed = 17;

Player.prototype.update = function(time, state, keys) {
  let xSpeed = 0;
  if (keys.ArrowLeft) xSpeed -= playerXSpeed;
  if (keys.ArrowRight) xSpeed += playerXSpeed;
  let pos = this.pos;
  let movedX = pos.plus(new Vec(xSpeed * time, 0));
  if (!state.level.touches(movedX, this.size, "wall")) {
    pos = movedX;
  }

  let ySpeed = this.speed.y + time * gravity;
  let movedY = pos.plus(new Vec(0, ySpeed * time));
  if (!state.level.touches(movedY, this.size, "wall")) {
    pos = movedY;
  } else if (keys.ArrowUp && ySpeed > 0) {
    ySpeed = -jumpSpeed;
  } else {
    ySpeed = 0;
   }
  return new Player(pos, new Vec(xSpeed, ySpeed));
};
```

水平运动根据左右箭头键的状态计算。当没有墙壁阻挡由这个运动产生的新位置时，就使用它。否则，保留旧位置。

垂直运动的原理类似，但必须模拟跳跃和重力。玩家的垂直速度（`ySpeed`）首先考虑重力而加速。

我们再次检查墙壁。如果我们不碰到任何一个，使用新的位置。如果存在一面墙，就有两种可能的结果。当按下向上的箭头，并且我们向下移动时（意味着我们碰到的东西在我们下面），将速度设置成一个相对大的负值。这导致玩家跳跃。否则，玩家只是撞到某物上，速度就被设定为零。

重力、跳跃速度和几乎所有其他常数，在游戏中都是通过反复试验来设定的。我测试了值，直到我找到了我喜欢的组合。

跟踪按键

对于这样的游戏，我们不希望按键在每次按下时生效。相反，我们希望只要按下了它们，他们的效果（移动球员的数字）就一直有效。

我们需要设置一个键盘处理器来存储左、右、上键的当前状态。我们调用`preventDefault`，防止按键产生页面滚动。

下面的函数接受一个按键名称数组，返回跟踪这些按键的当前位置的对象。并注册`"keydown"`和`"keyup"`事件，当事件对应的按键代码存在于其存储的按键代码集合中时，就更新对象。

```
function trackKeys(keys) {
  let down = Object.create(null);
  function track(event) {
    if (keys.includes(event.key)) {
      down[event.key] = event.type == "keydown";
      event.preventDefault();
    }
  }
  window.addEventListener("keydown", track);
  window.addEventListener("keyup", track);
  return down;
}

const arrowKeys =
  trackKeys(["ArrowLeft", "ArrowRight", "ArrowUp"]);
```

两种事件类型都使用相同的处理程序函数。该处理函数根据事件对象的type属性来确定是将按键状态修改为true（“keydown”）还是false（“keyup”）。

运行游戏

我们在第十四章中看到的`requestAnimationFrames`函数是一种产生游戏动画的好方法。但该函数的接口有点过于原始。该函数要求我们跟踪上次调用函数的时间，并在每一帧后再次调用`requestAnimationFrame`方法。

我们这里定义一个辅助函数来将这部分烦人的代码包装到一个名为`runAnimation`的简单接口中，我们只需向其传递一个函数即可，该函数的参数是一个时间间隔，并用于绘制一帧图像。当帧函数返回`false`时，整个动画停止。

```
function runAnimation(frameFunc) {
  let lastTime = null;
  function frame(time) {
    let stop = false;
    if (lastTime != null) {
      let timeStep = Math.min(time - lastTime, 100) / 1000;
      if (frameFunc(timeStep) === false) return;
    }
    lastTime = time;
    requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);
}
```

我们将每帧之间的最大时间间隔设置为 100 毫秒（十分之一秒）。当浏览器标签页或窗口隐藏时，`requestAnimationFrame` 调用会自动暂停，并在标签页或窗口再次显示时重新开始绘制动画。在本例中，`lastTime` 和 `time` 之差是隐藏页面的整个时间。一步一步地推进游戏看起来很傻，可能会造成奇怪的副作用，比如玩家从地板上掉下去。

该函数也会将时间单位转换成秒，相比于毫秒大家会更熟悉秒。

`runLevel` 函数的接受Level对象和显示对象的构造器，并返回一个 `Promise`。`runLevel` 函数（在 `document.body` 中）显示关卡，并使得用户通过该节点操作游戏。当关卡结束时（或胜或负），`runLevel` 会多等一秒（让用户看看发生了什么），清除关卡，并停止动画，如果我们指定了 `andThen` 函数，则 `runLevel` 会以关卡状态为参数调用该函数。

```
function runLevel(level, Display) {
  let display = new Display(document.body, level);
  let state = State.start(level);
  let ending = 1;
  return new Promise(resolve => {
    runAnimation(time => {
      state = state.update(time, arrowKeys);
      display.setState(state);
      if (state.status == "playing") {
        return true;
      } else if (ending > 0) {
        ending -= time;
        return true;
      } else {
        display.clear();
        resolve(state.status);
        return false;
      }
    });
  });
}
```

一个游戏是一个关卡序列。每当玩家死亡时就重新开始当前关卡。当完成关卡后，我们切换到下一关。我们可以使用下面的函数来完成该任务，该函数的参数为一个关卡平面图（字符串）数组和显示对象的构造器。

```
async function runGame(plans, Display) {
  for (let level = 0; level < plans.length;) {
    let status = await runLevel(new Level(plans[level]),
                                Display);
    if (status == "won") level++;
  }
  console.log("You've won!");
}
```

因为我们使 `runLevel` 返回 `Promise`，`runGame` 可以使用 `async` 函数编写，如第十一章中所见。它返回另一个 `Promise`，当玩家完成游戏时得到解析。

在本章的沙盒的 `GAME_LEVELS` 绑定中，有一组可用的关卡平面图。这个页面将它们提供给 `runGame`，启动实际的游戏：

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

习题

游戏结束

按照惯例，平台游戏中玩家一开始会有有限数量的生命，每死亡一次就扣去一条生命。当玩家生命耗尽时，游戏就从头开始了。

调整 `runGame` 来实现生命机制。玩家一开始会有 3 条生命。每次启动时输出当前生命数量（使用 `console.log`）。

```
<link rel="stylesheet" href="css/game.css">

<body>
<script>
  // The old runGame function. Modify it...
  async function runGame(plans, Display) {
    for (let level = 0; level < plans.length;) {
      let status = await runLevel(new Level(plans[level]),
                                  Display);
      if (status == "won") level++;
    }
    console.log("You've won!");
  }
  runGame(GAME_LEVELS, DOMDisplay);
</script>
</body>
```

暂停游戏

现在实现一个功能 —— 当用户按下 ESC 键时可以暂停或继续游戏。

我们可以修改 `runLevel` 函数，使用另一个键盘事件处理器来实现在玩家按下 ESC 键的时候中断或恢复动画。

乍看起来，`runAnimation` 无法完成该任务，但如果我们使用 `runLevel` 来重新安排调度策略，也是可以实现的。

当你完成该功能后，可以尝试加入另一个功能。我们现在注册键盘事件处理器的方法多少有点问题。现在 `arrows` 对象是一个全局绑定，即使游戏没有运行时，事件处理器也是有效的。我们称之为系统泄露。请扩展 `trackeys` ，提供一种方法来注销事件处理器，接着修改 `runLevel` 在启动游戏时注册事件处理器，并在游戏结束后注销事件处理器。

```
<link rel="stylesheet" href="css/game.css">

<body>
<script>
  // The old runLevel function. Modify this...
  function runLevel(level, Display) {
    let display = new Display(document.body, level);
    let state = State.start(level);
    let ending = 1;
    return new Promise(resolve => {
      runAnimation(time => {
        state = state.update(time, arrowKeys);
        display.setState(state);
        if (state.status == "playing") {
          return true;
        } else if (ending > 0) {
          ending -= time;
          return true;
        } else {
          display.clear();
          resolve(state.status);
          return false;
        }
      });
    });
  }
  runGame(GAME_LEVELS, DOMDisplay);
</script>
</body>
```

怪物

它是传统的平台游戏，里面有敌人，你可以跳到它顶上来打败它。这个练习要求你把这种角色类型添加到游戏中。

我们称之为怪物。怪物只能水平移动。你可以让它们朝着玩家的方向移动，或者像水平岩浆一样来回跳动，或者拥有你想要的任何运动模式。这个类不必处理掉落，但是它应该确保怪物不会穿过墙壁。

当怪物接触玩家时，效果取决于玩家是否跳到它们顶上。你可以通过检查玩家的底部是否接近怪物的顶部来近似它。如果是这样的话，怪物就消失了。如果没有，游戏就输了。

```
<link rel="stylesheet" href="css/game.css">
<style>.monster { background: purple }</style>

<body>
  <script>
    // Complete the constructor, update, and collide methods
    class Monster {
      constructor(pos, /* ... */) {}

      get type() { return "monster"; }

      static create(pos) {
        return new Monster(pos.plus(new Vec(0, -1)));
      }

      update(time, state) {}

      collide(state) {}
    }

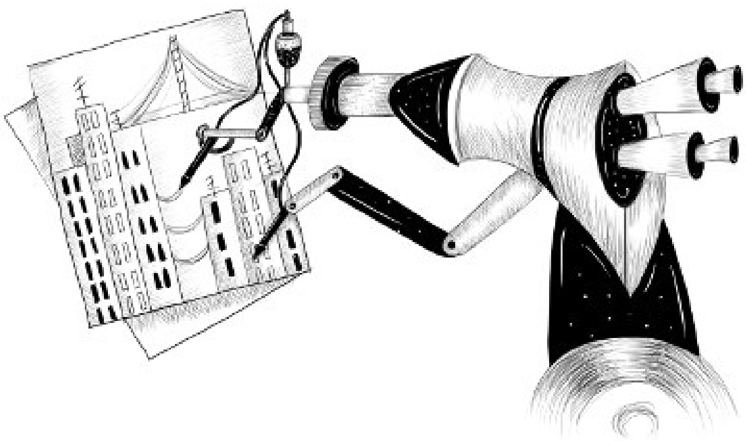
    Monster.prototype.size = new Vec(1.2, 2);

    levelChars["M"] = Monster;

    runLevel(new Level(`
..................................
.################################.
.#..............................#.
.#..............................#.
.#..............................#.
.#...........................o..#.
.#..@...........................#.
.##########..............########.
..........#..o..o..o..o..#........
..........#...........M..#........
..........################........
..................................
`), DOMDisplay);
  </script>
</body>
```

十七、在画布上绘图

> 原文：Drawing on Canvas
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 绘图就是欺骗。
>
> M.C. Escher，由 Bruno Ernst 在《The Magic Mirror of M.C. Escher》中引用



浏览器为我们提供了多种绘图方式。最简单的方式是用样式来规定普通 DOM 对象的位置和颜色。就像在上一章中那个游戏展示的，我们可以使用这种方式实现很多功能。我们可以为节点添加半透明的背景图片，来获得我们希望的节点外观。我们也可以使用`transform`样式来旋转或倾斜节点。

但是，在一些场景中，使用 DOM 并不符合我们的设计初衷。比如我们很难使用普通的 HTML 元素画出任意两点之间的线段这类图形。

这里有两种解决办法。第一种方法基于 DOM，但使用可缩放矢量图形（SVG，Scalable Vector Graphics）代替 HTML。我们可以将 SVG 看成文档标记方言，专用于描述图形而非文字。你可以在 HTML 文档中嵌入 SVG，还可以在`<img>`标签中引用它。

我们将第二种方法称为画布（canvas）。画布是一个能够封装图片的 DOM 元素。它提供了在空白的`html`节点上绘制图形的编程接口。SVG 与画布的最主要区别在于 SVG 保存了对于图像的基本信息的描述，我们可以随时移动或修改图像。

另外，画布在绘制图像的同时会把图像转换成像素（在栅格中的具有颜色的点）并且不会保存这些像素表示的内容。唯一的移动图形的方法就是清空画布（或者围绕着图形的部分画布）并在新的位置重画图形。

SVG

本书不会深入研究 SVG 的细节，但是我会简单地解释其工作原理。在本章的结尾，我会再次来讨论，对于某个具体的应用来说，我们应该如何权衡利弊选择一种绘图方式。

这是一个带有简单的 SVG 图片的 HTML 文档。

```
<p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
        stroke="blue" fill="none"/>
</svg>
```

`xmlns` 属性把一个元素（以及他的子元素）切换到一个不同的 XML 命名空间。这个由 `url` 定义的命名空间，规定了我们当前使用的语言。在 HTML 中不存在 `<circle>` 与 `<rect>` 标签，但这些标签在 SVG 中是有意义的，你可以通过这些标签的属性来绘制图像并指定样式与位置。

和 HTML 标签一样，这些标签会创建 DOM 元素，脚本可以和它们交互。例如，下面的代码可以把 `<circle>` 元素的颜色替换为青色。

```
let circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

`canvas` 元素

我们可以在 `<canvas>` 元素中绘制画布图形。你可以通过设置 `width` 与 `height` 属性来确定画布尺寸（单位为像素）。

新的画布是空的，意味着它是完全透明的，看起来就像文档中的空白区域一样。

`<canvas>` 标签允许多种不同风格的绘图。要获取真正的绘图接口，首先我们要创建一个能够提供绘图接口的方法的上下文（context）。目前有两种得到广泛支持的绘图接口：用于绘制二维图形的 `"2d"` 与通过openGL接口绘制三维图形的 `"webgl"` 。

本书只讨论二维图形，而不讨论 WebGL。但是如果你对三维图形感兴趣，我强烈建议大家自行深入研究 WebGL。它提供了非常简单的现代图形硬件接口，同时你也可以使用 JavaScript 来高效地渲染非常复杂的场景。

您可以用 `getContext` 方法在 `<canvas>` DOM 元素上创建一个上下文。

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  let canvas = document.querySelector("canvas");
  let context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

在创建完`context`对象之后，作为示例，我们画出一个红色矩形。该矩形宽 100 像素，高 50 像素，它的左上点坐标为(10,10)。

与 HTML（或者 SVG）相同，画布使用的坐标系统将`(0,0)`放置在左上角，并且`y`轴向下增长。所以`(10,10)`是相对于左上角向下并向右各偏移 10 像素的位置。

直线和平面

我们可以使用画布接口填充图形，也就是赋予某个区域一个固定的填充颜色或填充模式。我们也可以描边，也就是沿着图形的边沿画出线段。SVG 也使用了相同的技术。

`fillRect`方法可以填充一个矩形。他的输入为矩形框左上角的第一个`x`和`y`坐标，然后是它的宽和高。相似地，`strokeRect`方法可以画出一个矩形的外框。

两个方法都不需要其他任何参数。填充的颜色以及轮廓的粗细等等都不能由方法的参数决定（像你的合理预期一样），而是由上下文对象的属性决定。

设置`fillStyle`参数控制图形的填充方式。我们可以将其设置为描述颜色的字符串，使用 CSS 所用的颜色表示法。

`strokeStyle`属性的作用很相似，但是它用于规定轮廓线的颜色。线条的宽度由`lineWidth`属性决定。`lineWidth`的值都为正值。

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.strokeStyle = "blue";
  cx.strokeRect(5, 5, 50, 50);
  cx.lineWidth = 5;
  cx.strokeRect(135, 5, 50, 50);
</script>
```

当没有设置`width`或者`height`参数时，正如示例一样，画布元素的默认宽度为 300 像素，默认高度为 150 像素。

路径

路径是线段的序列。2D `canvas` 接口使用一种奇特的方式来描述这样的路径。路径的绘制都是间接完成的。我们无法将路径保存为可以后续修改并传递的值。如果你想修改路径，必须要调用多个方法来描述他的形状。

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  for (let y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>
```

本例创建了一个包含很多水平线段的路径，然后用 `stroke` 方法勾勒轮廓。每个线段都是由 `lineTo` 以当前位置为路径起点绘制的。除非调用了 `moveTo` ，否则这个位置通常是上一个线段的终点位置。如果调用了 `moveTo` ，下一条线段会从 `moveTo` 指定的位置开始。

当使用 `fill` 方法填充一个路径时，我们需要分别填充这些图形。一个路径可以包含多个图形，每个 `moveTo` 都会创建一个新的图形。但是在填充之前我们需要封闭路径（路径的起始节点与终止节点必须是同一个点）。如果一个路径尚未封闭，会出现一条从终点到起点的线段，然后才会填充整个封闭图形。

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
</script>
```

本例画出了一个被填充的三角形。注意只显示地画出了三角形的两条边。第三条从右下角回到上顶点的边是没有显示地画出，因而在勾勒路径的时候也不会存在。

你也可以使用 `closePath` 方法显示地通过增加一条回到路径起始节点的线段来封闭一个路径。这条线段在勾勒路径的时候将被显示地画出。

曲线

路径也可能会包含曲线。绘制曲线更加复杂。

`quadraticCurveTo` 方法绘制到某一个点的曲线。为了确定一条线段的曲率，需要设定一个控制点以及一个目标点。设想这个控制点会吸引这条线段，使其成为曲线。线段不会穿过控制点。但是，它起点与终点的方向会与两个点到控制点的方向平行。见下例：

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control=(60,10) goal=(90,90)
  cx.quadraticCurveTo(60, 10, 90, 90);
  cx.lineTo(60, 10);
  cx.closePath();
  cx.stroke();
</script>
```

我们从左到右绘制一个二次曲线，曲线的控制点坐标为 `(60,10)` ，然后画出两条穿过控制点并且回到线段起点的线段。绘制的结果类似一个星际迷航的图章。你可以观察到控制点的效果：从下端的角落里发出的线段朝向控制点并向他们的目标点弯曲。

`bezierCurve` （贝塞尔曲线）方法可以绘制一种类似的曲线。不同的是贝塞尔曲线需要两个控制点而不是一个，线段的每一个端点都需要一个控制点。下面是描述贝塞尔曲线的简单示例。

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control1=(10,10) control2=(90,10) goal=(50,90)
  cx.bezierCurveTo(10, 10, 90, 10, 50, 90);
  cx.lineTo(90, 10);
  cx.lineTo(10, 10);
  cx.closePath();
  cx.stroke();
</script>
```

两个控制点规定了曲线两个端点的方向。两个控制点相对两个端点的距离越远，曲线就会越向这个方向凸出。

由于我们没有明确的方法，来找出我们希望绘制图形所对应的控制点，所以这种曲线还是很难操控。有时候你可以通过计算得到他们，而有时候你只能通过不断的尝试来找到合适的值。

`arc` 方法是一种沿着圆的边缘绘制曲线的方法。 它需要弧的中心的一对坐标，半径，然后是起始和终止角度。

我们可以使用最后两个参数画出部分圆。角度是通过弧度来测量的，而不是度数。这意味着一个完整的圆拥有 `2π` 的弧度，或者 `2*Math.PI` （大约为 6.28）的弧度。弧度从圆心右边的点开始并以顺时针的方向计数。你可以以 0 起始并以一个比 `2π` 大的数值（比如 7）作为终止值，画出一个完整的圆。

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  // center=(50,50) radius=40 angle=0 to 7
  cx.arc(50, 50, 40, 0, 7);
  // center=(150,50) radius=40 angle=0 to ½π
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

上面这段代码绘制出的图形包含了一条从完整圆（第一次调用`arc`）的右侧到四分之一圆（第二次调用`arc`）的左侧的直线。`arc`与其他绘制路径的方法一样，会自动连接到上一个路径上。你可以调用`moveTo`或者开启一个新的路径来避免这种情况。

绘制饼状图

设想你刚刚从 EconomiCorp 获得了一份工作，并且你的第一个任务是画出一个描述其用户满意度调查结果的饼状图。`results`绑定包含了一个表示调查结果的对象的数组。

```
const results = [
  {name: "Satisfied", count: 1043, color: "lightblue"},
  {name: "Neutral", count: 563, color: "lightgreen"},
  {name: "Unsatisfied", count: 510, color: "pink"},
  {name: "No comment", count: 175, color: "silver"}
];
```

要想画出一个饼状图，我们需要画出很多个饼状图的切片，每个切片由一个圆弧与两条到圆心的线段组成。我们可以通过把一个整圆（`2π`）分割成以调查结果数量为单位的若干份，然后乘以做出相应选择的用户的个数来计算每个圆弧的角度。

```
<canvas width="200" height="200"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let total = results
    .reduce((sum, {count}) => sum + count, 0);
  // Start at the top
  let currentAngle = -0.5 * Math.PI;
  for (let result of results) {
    let sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.beginPath();
    // center=100,100, radius=100
    // from current angle, clockwise by slice's angle
    cx.arc(100, 100, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(100, 100);
    cx.fillStyle = result.color;
    cx.fill();
  }
</script>
```

但表格并没有告诉我们切片代表的含义，它毫无用处。因此我们需要将文字画在画布上。

文本

2D 画布的 `context` 对象提供了 `fillText` 方法和 `strokeText` 方法。第二个方法可以用于绘制字母轮廓，但通常情况下我们需要的是 `fillText` 方法。该方法使用当前的 `fillColor` 来填充特定文字的轮廓。

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.font = "28px Georgia";
  cx.fillStyle = "fuchsia";
  cx.fillText("I can draw text, too!", 10, 50);
</script>
```

你可以通过 `font` 属性来设定文字的大小，样式和字体。本例给出了一个字体的大小和字体族名称。也可以添加 `italic` 或者 `bold` 来选择样式。

传递给 `fillText` 和 `strokeText` 的后两个参数用于指定绘制文字的位置。默认情况下，这个位置指定了文字的字符基线（ `baseline` ）的起始位置，我们可以将其假想为字符所站立的位置，基线不考虑 `j` 或 `p` 字母中那些向下突出的部分。你可以设置 `textAlign` 属性（ `end` 或 `center` ）来改变起始点的水平位置，也可以设置 `textBaseline` 属性（ `top` 、 `middle` 或 `bottom` ）来设置基线的竖直位置。

在本章末尾的练习中，我们会回顾饼状图，并解决给饼状图分片标注的问题。

图像

计算机图形学领域经常将矢量图形和位图图形分开来讨论。本章一直在讨论第一种图形，即通过对图形的逻辑描述来绘图。而位图则相反，不需要设置实际图形，而是通过处理像素数据来绘制图像（光栅化的着色点）。

我们可以使用 `drawImage` 方法在画布上绘制像素值。此处的像素数值可以来自 `<img>` 元素，或者来自其他的画布。下例创建了一个独立的 `<img>` 元素，并且加载了一张图像文件。但我们无法马上使用该图片进行绘制，因为浏览器可能还没有完成图片的获取操作。为了处理这个问题，我们在图像元素上注册一个 `"load"` 事件处理程序并且在图片加载完之后开始绘制。

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/hat.png";
  img.addEventListener("load", () => {
    for (let x = 10; x < 200; x += 30) {
      cx.drawImage(img, x, 10);
    }
  });
</script>
```

默认情况下，`drawImage` 会根据原图的尺寸绘制图像。你也可以增加两个参数来设置不同的宽度和高度。

如果我们向 `drawImage` 函数传入 9 个参数，我们可以用其绘制出一张图片的某一部分。第二个到第五个参数表示需要拷贝的源图片中的矩形区域（`x`，`y` 坐标，宽度和高度），同时第六个到第九个参数给出了需要拷贝到的目标矩形的位置（在画布上）。



该方法可以用于在单个图像文件中放入多个精灵（图像单元）并画出你需要的部分。

我们可以改变绘制的人物造型，来展现一段看似人物在走动的动画。

`clearRect` 方法可以帮助我们在画布上绘制动画。该方法类似于 `fillRect` 方法，但是不同的是 `clearRect` 方法会将目标矩形透明化，并移除掉之前绘制的像素值，而不是着色。

我们知道每个精灵和每个子画面的宽度都是 24 像素，高度都是 30 像素。下面的代码装载了一幅图片并设置定时器（会重复触发的定时器）来定时绘制下一帧。

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    let cycle = 0;
    setInterval(() => {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
                   // source rectangle
                   cycle * spriteW, 0, spriteW, spriteH,
                   // destination rectangle
                   0,                0, spriteW, spriteH);
      cycle = (cycle + 1) % 8;
    }, 120);
  });
</script>
```

`cycle` 绑定用于记录角色在动画图像中的位置。每显示一帧，我们都要将 `cycle` 加 1，并通过取余数确保 `cycle` 的值在 0~7 这个范围内。我们随后使用该绑定计算精灵当前形象在图片中的 `x` 坐标。

变换

但是，如果我们希望角色可以向左走而不是向右走该怎么办？诚然，我们可以绘制另一组精灵，但我们也可以使用另一种方式在画布上绘图。

我们可以调用 `scale` 方法来缩放之后绘制的任何元素。该方法接受两个输入参数，第一个参数是水平缩放比例，第二个参数是竖直缩放比例。

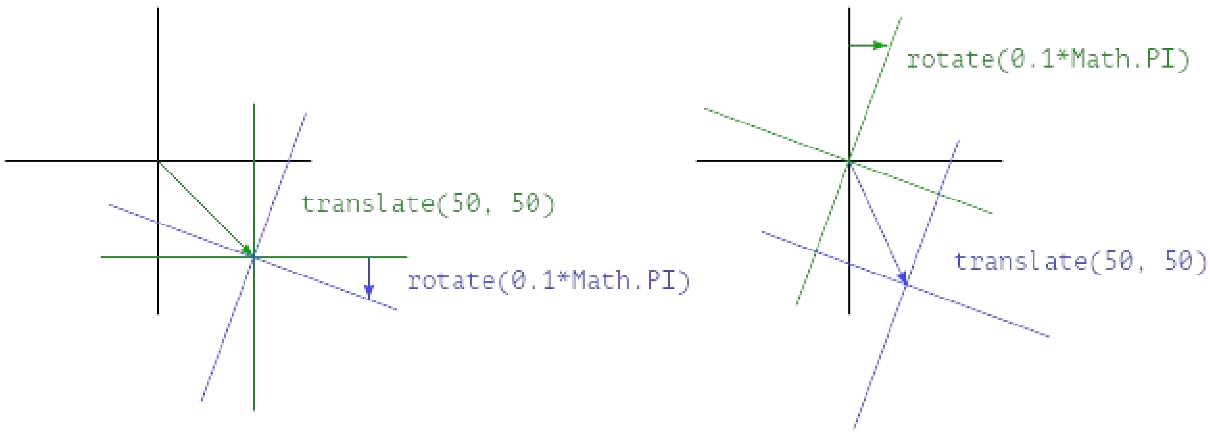
```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.scale(3, .5);
  cx.beginPath();
  cx.arc(50, 50, 40, 0, 7);
  cx.lineWidth = 3;
  cx.stroke();
</script>
```

因为调用了 `scale`，因此圆形长度变为原来的 3 倍，高度变为原来的一半。`scale` 可以调整图像所有特征，包括线宽、预定拉伸或压缩。如果将缩放值设置为负值，可以将图像翻转。由于翻转发生在坐标 `(0,0)` 处，这意味着也会同时反转坐标系的方向。当水平缩放 –1 时，在 `x` 坐标为 100 的位置画出的图形会绘制在缩放之前 `x` 坐标为 –100 的位置。

为了翻转一张图片，只是在 `drawImage` 之前添加 `cx.scale(-1,-1)` 是没用的，因为这样会将我们的图片移出到画布之外，导致图片不可见。为了避免这个问题，我们还需要调整传递给 `drawImage` 的坐标，将绘制图形的 `x` 坐标改为 –50 而不是 0。另一个解决方案是在缩放时调整坐标轴，这样代码就不需要知道整个画布的缩放的改变。

除了 `scale` 方法还有一些其他方法可以影响画布里坐标系统的方法。你可以使用 `rotate` 方法旋转绘制完的图形，也可以使用 `translate` 方法移动图形。毕竟有趣但也容易引起误解的是这些变换以栈的方式工作，也就是说每个变换都会作用于前一个变换的结果之上。

如果我们沿水平方向将画布平移两次，每次移动 10 像素，那么所有的图形都会在右方 20 像素的位置重新绘制。如果我们先把坐标系的原点移动到 `(50, 50)` 的位置，然后旋转 20 度（大约 `0.1π` 弧度），此次的旋转会围绕点 `(50,50)` 进行。

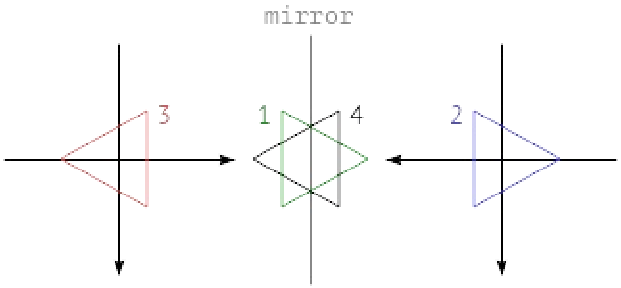


但是如果我们先旋转 20 度，然后平移原点到 `(50,50)`，此次的平移会发生在已经旋转过的坐标系中，因此会有不同的方向。变换发生顺序会影响最后的结果。

我们可以使用下面的代码，在指定的 `x` 坐标处竖直反转一张图片。

```
function flipHorizontally(context, around) {
  context.translate(around, 0);
  context.scale(-1, 1);
  context.translate(-around, 0);
}
```

我们先把 `y` 轴移动到我们希望镜像所在的位置，然后进行镜像翻转，最后把 `y` 轴移动到被翻转的坐标系当中相应的位置。下面的图片解释了以上代码是如何工作的：



上图显示了通过中线进行镜像翻转前后的坐标系。对三角形编号来说明每一步。如果我们在 `x` 坐标为正值的位置绘制一个三角形，默认情况下它会出现在图中三角形 1 的位置。调用 `filpHorizontally` 首先做一个向右的平移，得到三角形 2。然后将其翻转到三角形 3 的位置。这不是它的根据给定的中线翻转之后应该在的最终位置。第二次调用 `translate` 方法解决了这个问题。它“去除”了最初的平移的效果，并且使三角形 4 变成我们希望的效果。

我们可以沿着特征的竖直中心线翻转整个坐标系，这样就可以画出位置为 `(100,0)` 处的镜像特征。

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
                 100, 0, spriteW, spriteH);
  });
</script>
```

存储与清除图像的变换状态

图像变换的效果会保留下来。我们绘制出一次镜像特征后，绘制其他特征时都会产生镜像效果，这可能并不方便。

对于需要临时转换坐标系统的函数来说，我们经常需要保存当前的信息，画一些图，变换图像然后重新加载之前的图像。首先，我们需要将当前函数调用的所有图形变换信息保存起来。接着，函数完成其工作，并添加更多的变换。最后我们恢复之前保存的变换状态。

2D 画布上下文的 `save` 与 `restore` 方法执行这个变换管理。这两个方法维护变换状态堆栈。`save` 方法将当前状态压到堆栈中，`restore` 方法将堆栈顶部的状态弹出，并将该状态作为当前 `context` 对象的状态。

下面示例中的 `branch` 函数首先修改变换状态，然后调用其他函数（本例中就是该函数自身）继续在特定变换状态中进行绘图。

这个方法通过画出一条线段，并把坐标系的中心移动到线段的端点，然后调用自身两次，先向左旋转，接着向右旋转，来画出一个类似树一样的图形。每次调用都会减少所画分支的长度，当长度小于 8 的时候递归结束。

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>
```

如果没有调用 `save` 与 `restore` 方法，第二次递归调用 `branch` 将会在第一次调用的位置结束。它不会与当前的分支相连接，而是更加靠近中心偏右第一次调用所画出的分支。结果图像会很有趣，但是它肯定不是一棵树。

回到游戏

我们现在已经了解了足够多的画布绘图知识，我们已经可以使用基于画布的显示系统来改造前面几章中开发的游戏了。新的界面不会再是一个个色块，而使用 `drawImage` 来绘制游戏中元素对应的图片。

我们定义了一种对象类型，叫做 `CanvasDisplay`，支持第 14 章中的 `DOMDisplay` 的相同接口，也就是 `setState` 方法与 `clear` 方法。

这个对象需要比 `DOMDisplay` 多保存一些信息。该对象不仅需要使用 DOM 元素的滚动位置，还需要追踪自己的视口（viewport）。视口会告诉我们目前处于哪个关卡。最后，该对象会保存一个 `filpPlayer` 属性，确保即便玩家站立不动时，它面朝的方向也会与上次移动所面向的方向一致。

```
class CanvasDisplay {
  constructor(parent, level) {
    this.canvas = document.createElement("canvas");
    this.canvas.width = Math.min(600, level.width * scale);
    this.canvas.height = Math.min(450, level.height * scale);
    parent.appendChild(this.canvas);
    this.cx = this.canvas.getContext("2d");

    this.flipPlayer = false;

    this.viewport = {
      left: 0,
      top: 0,
      width: this.canvas.width / scale,
      height: this.canvas.height / scale
    };
  }

  clear() {
    this.canvas.remove();
  }
}
```

`setState` 方法首先计算一个新的视口，然后在适当的位置绘制游戏场景。

```
CanvasDisplay.prototype.setState = function(state) {
  this.updateViewport(state);
  this.clearDisplay(state.status);
  this.drawBackground(state.level);
  this.drawActors(state.actors);
};
```

与 `DOMDisplay` 相反，这种显示风格确实必须在每次更新时重新绘制背景。 因为画布上的形状只是像素，所以在我们绘制它们之后，没有什么好方法来移动它们（或将它们移除）。 更新画布显示的唯一方法，是清除它并重新绘制场景。 我们也可能发生了滚动，这要求背景处于不同的位置。

`updateViewport` 方法与 `DOMDisplay` 的 `scrollPlayerintoView` 方法相似。它检查玩家是否过于接近屏幕的边缘，并且当这种情况发生时移动视口。

```
CanvasDisplay.prototype.updateViewport = function(state) {
  let view = this.viewport, margin = view.width / 3;
  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5));

  if (center.x < view.left + margin) {
    view.left = Math.max(center.x - margin, 0);
  } else if (center.x > view.left + view.width - margin) {
    view.left = Math.min(center.x + margin - view.width,
                         state.level.width - view.width);
  }
  if (center.y < view.top + margin) {
    view.top = Math.max(center.y - margin, 0);
  } else if (center.y > view.top + view.height - margin) {
    view.top = Math.min(center.y + margin - view.height,
                        state.level.height - view.height);
  }
};
```

对`Math.max`和`Math.min`的调用保证了视口不会显示当前这层之外的物体。`Math.max(x,0)保证了结果数值不会小于 0。同样地，` Math.min`保证了数值保持在给定范围内。

在清空图像时，我们依据游戏是获胜（明亮的颜色）还是失败（灰暗的颜色）来使用不同的颜色。

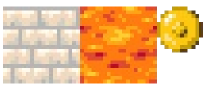
```
CanvasDisplay.prototype.clearDisplay = function(status) {
  if (status == "won") {
    this.cx.fillStyle = "rgb(68, 191, 255)";
  } else if (status == "lost") {
    this.cx.fillStyle = "rgb(44, 136, 214)";
  } else {
    this.cx.fillStyle = "rgb(52, 166, 251)";
  }
  this.cx.fillRect(0, 0,
                   this.canvas.width, this.canvas.height);
};
```

要画出一个背景，我们使用来自上一节的`touches`方法中的相同技巧，遍历在当前视口中可见的所有瓦片。

```
let otherSprites = document.createElement("img");
otherSprites.src = "img/sprites.png";

CanvasDisplay.prototype.drawBackground = function(level) {
  let {left, top, width, height} = this.viewport;
  let xStart = Math.floor(left);
  let xEnd = Math.ceil(left + width);
  let yStart = Math.floor(top);
  let yEnd = Math.ceil(top + height);
  for (let y = yStart; y < yEnd; y++) {
    for (let x = xStart; x < xEnd; x++) {
      let tile = level.rows[y][x];
      if (tile == "empty") continue;
      let screenX = (x - left) * scale;
      let screenY = (y - top) * scale;
      let tileX = tile == "lava" ? scale : 0;
      this.cx.drawImage(otherSprites,
                        tileX,         0, scale, scale,
                        screenX, screenY, scale, scale);
    }
  }
};
```

非空的瓦片是使用`drawImage`绘制的。`otherSprites`包含了描述除了玩家之外需要用到的图片。它包含了从左到右的墙上的瓦片，火山岩瓦片以及精灵硬币。



背景瓦片是`20×20`像素的，因为我们将要用到`DOMDisplay`中的相同比例。因此，火山岩瓦片的偏移是 20，墙面的偏移是 0。

我们不需要等待精灵图片加载完成。调用`drawImage`时使用一幅并未加载完毕的图片不会有任何效果。因为图片仍然在加载当中，我们可能无法正确地画出游戏的前几帧。但是这不是一个严重的问题，因为我们持续更新荧幕，正确的场景会在加载完毕之后立即出现。

前面展示过的走路的特征将会被用来代替玩家。绘制它的代码需要根据玩家的当前动作选择正确的动作和方向。前 8 个子画面包含一个走路的动画。当玩家沿着地板移动时，我们根据当前时间把他围起来。我们希望每 60 毫秒切换一次帧，所以时间先除以 60。当玩家站立不动时，我们画出第九张子画面。当竖直方向的速度不为 0，从而被判断为跳跃时，我们使用第 10 张，也是最右边的子画面。

因为子画面宽度为 24 像素而不是 16 像素，会稍微比玩家的对象宽，这时为了腾出脚和手的空间，该方法需要根据某个给定的值（`playerXOverlap`）调整`x`坐标的值以及宽度值。

```
let playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
const playerXOverlap = 4;

CanvasDisplay.prototype.drawPlayer = function(player, x, y,
                                              width, height){
  width += playerXOverlap * 2;
  x -= playerXOverlap;
  if (player.speed.x != 0) {
    this.flipPlayer = player.speed.x < 0;
  }

  let tile = 8;
  if (player.speed.y != 0) {
    tile = 9;
  } else if (player.speed.x != 0) {
    tile = Math.floor(Date.now() / 60) % 8;
  }

  this.cx.save();
  if (this.flipPlayer) {
    flipHorizontally(this.cx, x + width / 2);
  }
  let tileX = tile * width;
  this.cx.drawImage(playerSprites, tileX, 0, width, height,
                                   x,     y, width, height);
  this.cx.restore();
};
```

`drawPlayer`方法由`drawActors`方法调用，该方法负责画出游戏中的所有角色。

```
CanvasDisplay.prototype.drawActors = function(actors) {
  for (let actor of actors) {
    let width = actor.size.x * scale;
    let height = actor.size.y * scale;
    let x = (actor.pos.x - this.viewport.left) * scale;
    let y = (actor.pos.y - this.viewport.top) * scale;
    if (actor.type == "player") {
      this.drawPlayer(actor, x, y, width, height);
    } else {
      let tileX = (actor.type == "coin" ? 2 : 1) * scale;
      this.cx.drawImage(otherSprites,
                        tileX, 0, width, height,
                        x,     y, width, height);
    }
  }
};
```

当需要绘制一些非玩家元素时，我们首先检查它的类型，来找到与正确的子画面的偏移值。熔岩瓷砖出现在偏移为 20 的子画面，金币的子画面出现在偏移值为 40 的地方（放大了两倍）。

当计算角色的位置时，我们需要减掉视口的位置，因为 `(0,0)` 在我们的画布坐标系中代表着视口层面的左上角，而不是该关卡的左上角。我们也可以使用 `translate` 方法，这样可以作用于所有元素。

这个文档将新的显示屏插入 `runGame` 中：

```
<body>
  <script>
    runGame(GAME_LEVELS, CanvasDisplay);
  </script>
</body>
```

选择图像接口

所以当你需要在浏览器中绘图时，你都可以选择纯粹的 HTML、SVG 或画布。没有唯一的最适合的且在所有动画中都是最好的方法。每个选择都有它的利与弊。

单纯的 HTML 的优点是简单。它也可以很好地与文字集成使用。SVG 与画布都可以允许你绘制文字，但是它们不会只通过一行代码来帮助你放置 `text` 或者包装它，在一个基于 HTML 的图像中，包含文本块更加简单。

SVG 可以被用来制造可以任意缩放而仍然清晰的图像。与 HTML 相反，它实际上是为绘图而设计的，因此更适合于此目的。

SVG 与 HTML 都会构建一个新的数据结构（DOM），它表示你的图片。这使得在绘制元素之后对其进行修改更为可能。如果你需要重复的修改在一张大图片中的一小部分，来对用户的动作进行响应或者作为动画的一部分时，在画布里做这件事情将会极其的昂贵。DOM 也可以允许我们在图片上的每一个元素（甚至在 SVG 画出的图形上）注册鼠标事件的处理器。在画布里则实现不了。

但是画布的基于像素的方法在需要绘制大量的微小元素时会有优势。它不会构建新的数据结构而是仅仅重复的在同一个像素上绘制，这使得画布在每个图形上拥有更低的消耗。

有一些效果，像在逐像素的渲染一个场景（比如，使用光线追踪）或者使用 javaScript 对一张图片进行后加工（虚化或者扭曲），只能通过基于像素的技术来进行真实的处理。在某些情况下，你可能想要将这些技术整合起来使用。比如，你可能用 SVG 或者画布画出一个图形，但是通过将一个 HTML 元素放在图片的顶端来展示像素信息。

对于一些要求低的程序来说，选择哪个接口并没有什么太大的区别。因为不需要绘制文字，处理鼠标交互或者处理大量的元素。我们在本章为游戏构建的显示屏，可以通过使用三种图像技术中的任意一种来实现。

本章小结

在本章中，我们讨论了在浏览器中绘制图形的技术，重点关注了`<canvas>`元素。

一个`canvas`节点代表了我们的程序可以绘制在文档中的一片区域。这个绘图动作是通过一个由`getContext`方法创建的绘图上下文对象完成的。

2D 绘图接口允许我们填充或者拉伸各种各样的图形。这个上下文的`fillStyle`属性决定了图形的填充方式。`strokeStyle`和`lineWidth`属性用来控制线条的绘制方式。

矩形与文字可以通过使用一个简单的方法调用来绘制。采用`fillRect`和`strokeRect`方法绘制矩形，同时采用`fillText`和`strokeText`方法绘制文字。要创建一个自定义的图形，我们必须首先建立一个路径。

调用`beginPath`会创建一个新的路径。很多其他的方法可以向当前的路径添加线条和曲线。比如，`lineTo`方法可以添加一条直线。当一条路径画完时，它可以被`fill`方法填充或者被`stroke`方法勾勒轮廓。

从一张图片或者另一个画布上移动像素到我们的画布上可以用`drawImage`方法实现。默认情况下，这个方法绘制了整个原图像，但是通过给它更多的参数，你可以拷贝一张图片的某一个特定的区域。我们在游戏中使用了这项技术，从包括许多动作的图像中拷贝出游戏角色的单个独立动作。

图形变换允许你向多个方向绘制图片。2D 绘制上下文拥有一个当前的可以通过`translate`、`scale`与`rotate`进行变换。这些会影响所有的后续的绘制操作。一个变换的状态可以通过`save`方法来保存，通过`restore`方法来恢复。

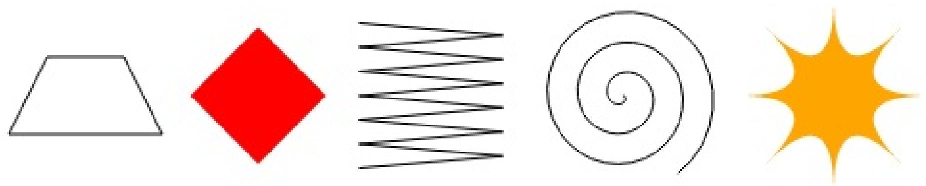
在一个画布上展示动画时，`clearRect`方法可以用来在重绘之前清除画布的某一部分。

习题

形状

编写一个程序，在画布上画出下面的图形。

1. 一个梯形（一个在一边比较长的矩形）
2. 一个红色的钻石（一个矩形旋转45度角）
3. 一个锯齿线
4. 一个由 100 条直线线段构成的螺旋
5. 一个黄色的星星



当绘制最后两个图形时，你可以参考第 14 章中的 `Math.cos` 和 `Math.sin` 的解释，它描述了如何使用这两个函数获得圆上的坐标。

建议你为每一个图形创建一个方法，传入坐标信息，以及其他的一些参数，比如大小或者点的数量。另一种方法，可以在你的代码中硬编码，会使得你的代码变得难以阅读和修改。

```
<canvas width="600" height="200"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");

  // Your code here.
</script>
```

饼状图

在本章的前部分，我们看到一个绘制饼状图的样例程序。修改这个程序，使得每个部分的名字可以被显示在相应的切片旁边。试着找到一个合适的方法来自动放置这些文字，同时也可以适用于其他数据。你可以假设分类大到足以为标签留出空间。

你可能还会需要 `Math.sin` 和 `Math.cos` 方法，像第 14 章描述的一样。

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let total = results
    .reduce((sum, {count}) => sum + count, 0);
  let currentAngle = -0.5 * Math.PI;
  let centerX = 300, centerY = 150;

  // Add code to draw the slice labels in this loop.
  results.forEach(function(result) {
  for (let result of results) {
    let sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.arc(centerX, centerY, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(centerX, centerY);
    cx.fillStyle = result.color;
    cx.fill();
  }
</script>
```

弹力球

使用在第 14 章和第 16 章出现的 `requestAnimationFrame` 方法画出一个装有弹力球的盒子。这个球匀速运动并且当撞到盒子的边缘的时候反弹。

```
<canvas width="400" height="400"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");

  let lastTime = null;
  function frame(time) {
    if (lastTime != null) {
      updateAnimation(Math.min(100, time - lastTime) / 1000);
    }
    lastTime = time;
    requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);

  function updateAnimation(step) {
    // Your code here.
  }
</script>
```

预处理镜像

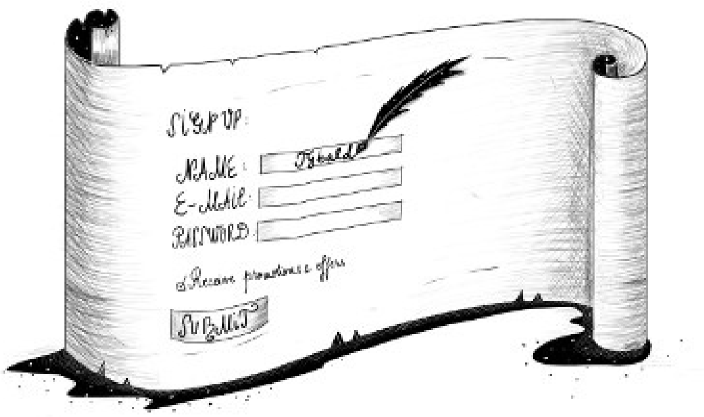
当进行图形变换时，绘制位图图像会很慢。每个像素的位置和大小都必须进行变换，尽管将来浏览器可能会更加聪明，但这会导致绘制位图所需的时间显着增加。

在一个像我们这样的只绘制一个简单的子画面图像变换的游戏中，这个不是问题。但是如果我们需要绘制成百上千的角色或者爆炸产生的旋转粒子时，这将会成为一个问题。

思考一种方法来允许我们不需要加载更多的图片文件就可以画出一个倒置的角色，并且不需要在每一帧调用`drawImage`方法。

十八、HTTP 和表单

> 原文：HTTP and Forms
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> 通信在实质上必须是无状态的，从客户端到服务器的每个请求都必须包含理解请求所需的所有信息，并且不能利用服务器上存储的任何上下文。
>
> Roy Fielding，《Architectural Styles and the Design of Network-based Software Architectures》



我们曾在第 13 章中提到过超文本传输协议（HTTP），万维网中通过该协议进行数据请求和传输。在本章中会对该协议进行详细介绍，并解释浏览器中 JavaScript 访问 HTTP 的方式。

协议

当你在浏览器地址栏中输入`eloquentjavascript.net/18_http.html`时，浏览器会首先找到和`eloquentjavascript.net`相关的服务器的地址，然后尝试通过 80 端口建立 TCP 连接，其中 80 端口是 HTTP 的默认通信端口。如果该服务器存在并且接受了该连接，浏览器可能发送如下内容。

```
GET /18_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Your browser's name
```

然后服务器会通过同一个链接返回如下内容。

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

<!doctype html>
... the rest of the document
```

浏览器会选取空行之后的响应部分，也就是正文（不要与 HTML `<body>` 标签混淆），并将其显示为 HTML 文档。

由客户端发出的信息叫作请求。请求的第一行如下。

```
GET /17_http.html HTTP/1.1
```

请求中的第一个单词是请求方法。`GET` 表示我们希望得到一个我们指定的资源。其他常用方式还有 `DELETE`，用于删除一个资源；`PUT` 用于替换资源；`POST` 用于发送消息。需要注意的是服务器并不需要处理所有收到的请求。如果你随机访问一个网站并请求删除主页，服务器很有可能会拒绝你的请求。

方法名后的请求部分是所请求的资源的路径。在最简单的情况下，一个资源只是服务器中的一个文件。不过，协议并没有要求资源一定是实际文件。一个资源可以是任何可以像文件一样传输的东西。很多服务器会实时地生成这些资源。例如，如果你打开 `github.com/marijnh`，服务器会在数据库中寻找名为 `marijnh` 的用户，如果找到了则会为该用户的生成介绍页面。

请求的第一行中位于资源路径后面的 `HTTP/1.1` 用来表明所使用的 HTTP 协议的版本。

在实践中，许多网站使用 HTTP v2，它支持与版本 1.1 相同的概念，但是要复杂得多，因此速度更快。 浏览器在与给定服务器通信时，会自动切换到适当的协议版本，并且无论使用哪个版本，请求的结果都是相同的。 由于 1.1 版更直接，更易于使用，因此我们将专注于此。

服务器的响应也是以版本号开始的。版本号后面是响应状态，首先是一个三位的状态码，然后是一个可读的字符串。

```
HTTP/1.1 200 OK
```

以 2 开头的状态码表示请求成功。以 4 开头的状态码表示请求中有错误。404 是最著名的 HTTP 状态码了，表示找不到资源。以 5 开头的状态码表示服务器端出现了问题，而请求没有问题。

请求或响应的第一行后可能会有任意个协议头，多个形如 `name: value` 的行表明了和请求或响应相关的更多信息。这些是示例响应中的头信息。

```
Content-Length: 65585
Content-Type: text/html
Last-Modified: Thu, 04 Jan 2018 14:05:30 GMT
```

这些信息说明了响应文档的大小和类型。在这个例子中，响应是一个 65585 字节的 HTML 文档，同时也说明了该文档最后的更改时间。

多数大多数协议头，客户端或服务器可以自由决定需要在请求或响应中包含的协议头，不过也有一些协议头是必需的。例如，指明主机名的 `Host` 头在请求中是必须的，因为一个服务器可能在一个 IP 地址下有多个主机名服务，如果没有 `Host` 头，服务器则无法判断客户端尝试请求哪个主机。

请求和响应可能都会在协议头后包含一个空行，后面则是消息体，包含所发送的数据。`GET` 和 `DELETE` 请求不单独发送任何数据，但 `PUT` 和 `POST` 请求则会。同样地，一些响应类型（如错误响应）不需要有消息体。

浏览器和 HTTP

正如上例所示，当我们在浏览器地址栏输入一个 URL 后浏览器会发送一个请求。当 HTML 页面中包含有其他的文件，例如图片和 JavaScript 文件时，浏览器也会一并获取这些资源。

一个较为复杂的网站通常都会有 10 到 200 个不等的资源。为了可以很快地取得这些资源，浏览器会同时发送多个 `GET` 请求，而不是一次等待一个请求。此类文档都是通过 `GET` 方法来获取的。

HTML页面可能包含表单，用户可以在表单中填入一些信息然后由浏览器将其发送到服务器。如下是一个表单的例子。

```
<form method="GET" action="example/message.html">
  <p>Name: <input type="text" name="name"></p>
  <p>Message:<br><textarea name="message"></textarea></p>
  <p><button type="submit">Send</button></p>
</form>
```

这段代码描述了一个有两个输入字段的表单：较小的输入字段要求用户输入姓名，较大的要求用户输入一条消息。当点击发送按钮时，表单就提交了，这意味着其字段的内容被打包到 HTTP 请求中，并且浏览器跳转到该请求的结果。

当 `<form>` 元素的 `method` 属性是 `GET`（或省略）时，表单中的信息将作为查询字符串添加到 `action` URL 的末尾。浏览器可能会向此 URL 发出请求：

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

问号表示路径的末尾和查询字符串的起始。后面是多个名称和值，这些名称和值分别对应 `form` 输入字段中的 `name` 属性和这些元素的内容。`&` 字符用来分隔不同的名称对。

在这个 URL 中，经过编码的消息实际原本是 `"Yes?"` ，只不过浏览器用奇怪的代码替换了问号。我们必须替换掉请求字符串中的一些字符。使用 `%3F` 替换的问号就是其中之一。这样看，似乎有一个不成文的规定，每种格式都会有自己的转义字符。这里的编码格式叫作 URL 编码，使用一个百分号和16进制的数字来对字符进行编码。在这个例子中，3F（十进制为 63）是问号字符的编码。JavaScript 提供了 `encodeURIComponent` 和 `decodeURIComponent` 函数来按照这种格式进行编码和解码。

```
console.log(encodeURIComponent("Yes?"));
// → Yes%3F
console.log(decodeURIComponent("Yes%3F"));
// → Yes?
```

如果我们将本例 HTML 表单中的 `method` 属性更改为 `POST` ，则浏览器会使用 `POST` 方法发送该表单，并将请求字符串放到请求正文中，而不是添加到 URL 中。

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F
```

`GET` 请求应该用于没有副作用的请求，而仅仅是询问信息。 可以改变服务器上的某些内容的请求，例如创建一个新帐户或发布消息，应该用其他方法表示，例如 `POST` 。 诸如浏览器之类的客户端软件，知道它不应该盲目地发出 `POST` 请求，但通常会隐式地发出 `GET` 请求 - 例如预先获取一个它认为用户很快需要的资源。

我们将在本章后面的回到表单，以及如何与 JavaScript 交互。

Fetch

浏览器 JavaScript 可以通过 `fetch` 接口生成 HTTP 请求。 由于它比较新，所以它很方便地使用了 `Promise` （这在浏览器接口中很少见）。

```
fetch("example/data.txt").then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```

调用 `fetch` 返回一个 `Promise` ，它解析为一个 `Response` 对象，该对象包含服务器响应的信息，例如状态码和协议头。 协议头被封装在类 `Map` 的对象中，该对象不区分键（协议头名称）的大小写，因为协议头名称不应区分大小写。 这意味

着 `header.get("Content-Type")` 和 `headers.get("content-TYPE")` 将返回相同的值。

请注意，即使服务器使用错误代码进行响应，由 `fetch` 返回的 `Promise` 也会成功解析。 如果存在网络错误或找不到请求的服务器，它也可能被拒绝。

`fetch` 的第一个参数是请求的 URL。 当该 URL 不以协议名称（例如 `http:` ）开头时，它被视为相对路径，这意味着它解释为相对于当前文档的路径。 当它以斜线（ `/` ）开始时，它将替换当前路径，即服务器名称后面的部分。 否则，当前路径直到并包括最后一个斜杠的部分，放在相对 URL 前面。

为了获取响应的实际内容，可以使用其 `text` 方法。 由于初始 `Promise` 在收到响应头文件后立即解析，并且读取响应正文可能需要一段时间，这又会返回一个 `Promise` 。

```
fetch("example/data.txt")
  .then(resp => resp.text())
  .then(text => console.log(text));
// → This is the content of data.txt
```

有一种类似的方法，名为 `json` ，它返回一个 `Promise` ，它将解析为，将正文解析为 JSON 时得到的值，或者不是有效的 JSON，则被拒绝。

默认情况下， `fetch` 使用 `GET` 方法发出请求，并且不包含请求正文。 你可以通过传递一个带有额外选项的对象作为第二个参数，来进行不同的配置。 例如，这个请求试图删除 `example/data.txt` 。

```
fetch("example/data.txt", {method: "DELETE"}).then(resp => {
  console.log(resp.status);
  // → 405
});
```

405 状态码意味着“方法不允许”，这是 HTTP 服务器说“我不能这样做”的方式。

为了添加一个请求正文，你可以包含 `body` 选项。 为了设置标题，存在 `headers` 选项。 例如，这个请求包含 `Range` 协议，它指示服务器只返回一部分响应。

```
fetch("example/data.txt", {headers: {Range: "bytes=8-19"}})
  .then(resp => resp.text())
  .then(console.log);
// → the content
```

浏览器将自动添加一些请求头，例如 `Host` 和服务器需要的协议头，来确定正文的大小。 但是对于包含认证信息或告诉服务器想要接收的文件格式，添加自己的协议头通常很有用。

HTTP 沙箱

在网页脚本中发出 HTTP 请求，再次引发了安全性的担忧。 控制脚本的人的兴趣可能不同于正在运行的计算机的所有者。 更具体地说，如果我访问 `themafia.org` ，我不希望其脚本能够使用来自我的浏览器的身份向 `mybank.com` 发出请求，并且下令将我所有的钱转移到某个随机帐户。

出于这个原因，浏览器通过禁止脚本向其他域（如 `themafia.org` 和 `mybank.com` 等名称）发送 HTTP 请求来保护我们。

在构建希望因合法原因访问多个域的系统时，这可能是一个恼人的问题。 幸运的是，服务器可以在响应中包含这样的协议头，来明确地向浏览器表明，请求可以来自另一个域：

```
Access-Control-Allow-Origin: *
```

运用 HTTP

当构建一个需要让浏览器（客户端）的 JavaScript 程序和服务器端的程序进行通信的系统时，有一些不同的方式可以实现这个功能。

一个常用的方法是远程过程调用，通信遵从正常的方法调用方式，不过调用的方法实际运行在另一台机器中。调用包括向服务器发送包含方法名和参数的请求。响应的结果则包括函数的返回值。

当考虑远程过程调用时，HTTP 只是通信的载体，并且你很可能会写一个抽象层来隐藏细节。

另一个方法是使用一些资源和 HTTP 方法来建立自己的通信。不同于远程调用方法 `addUser` ，你需要发送一个 `PUT` 请求到 `users/larry` ，不同于将用户属性进行编码后作为参数传递，你定义了一个 JSON 文档格式（或使用一种已有的格式）来展示一个用户。 `PUT` 请求的正文则只是这样的一个用来建立新资源的文档。由 `GET` 方法获取的资源则是自愿的 URL（例如， `/users/larry` ），该 URL 返回代表这个资源的文档。

第二种方法使用了 HTTP 的一些特性，所以使得整体更简洁。例如对于资源缓存的支持（在客户端存一份副本用于快速访问）。HTTP 中使用的概念设计良好，可以提供一组有用的原则来设计服务器接口。

安全和 HTTPS

通过互联网传播的数据，往往走过漫长而危险的道路。 为了到达目的地，它必须跳过任何东西，从咖啡店的 Wi-Fi 到由各个公司和国家管理的网络。 在它的路线上的任何位置，它都可能被探测或者甚至被修改。

如果对某件事保密是重要的，例如你的电子邮件帐户的密码，或者它到达目的地而未经修改是重要的，例如帐户号码，你使用它在银行网站上转账，纯 HTTP 就不够好了。

安全的 HTTP 协议，其 URL 以 `https://` 开头，是一种难以阅读和篡改的，HTTP 流量的封装方式。 在交换数据之前，客户端证实该服务器是它所声称的东西，通过要求它证明，它具有由浏览器承认的证书机构所颁发的证书。 接下来，通过连接传输的所有数据，都将以某种方式加密，它应该防止窃听和篡改。

因此，当 HTTPS 正常工作时，它可以阻止某人冒充你想要与之通话的网站，以及某人窥探你的通信。 这并不完美，由于伪造或被盗的证书和损坏的软件，存在各种 HTTPS 失败的事故，但它比纯 HTTP 更安全。

表单字段

表单最初是为 JavaScript 之前的网页设计的，允许网站通过 HTTP 请求发送用户提交的信息。 这种设计假定与服务器的交互，总是通过导航到新页面实现。

但是它们的元素是 DOM 的一部分，就像页面的其他部分一样，并且表示表单字段的 DOM 元素，支持许多其他元素上不存在的属性和事件。 这些使其可以使用 JavaScript 程序检查和控制这些输入字段，以及可以执行一些操作，例如向表单添加新功能，或在 JavaScript 应用程序中使用表单和字段作为积木。

一个网页表单在其 `<form>` 标签中包含若干个输入字段。HTML 允许多个的不同风格的输入字段，从简单的开关选择框到下拉菜单和进行输入的字段。本书不会全面的讨论每一个输入字段类型，不过我们会先大概讲述一下。

很多字段类型都使用 `<input>` 标签。标签的 `type` 属性用来选择字段的种类，下面是一些常用的 `<input>` 类型。

- `text`：一个单行的文本输入框。
- `password`：和 `text` 相同但隐藏了输入内容。
- `checkbox`：一个复选框。
- `radio`：一个多选择字段中的一个单选框。
- `file`：允许用户从本机选择文件上传。

表单字段并不一定要出现在 `<form>` 标签中。你可以把表单字段放置在一个页面的任何地方。但这样不带表单的字段不能被提交（一个完整的表单才可以），当需要和 JavaScript 进行响应时，我们通常也不希望按常规的方式提交表单。

```
<p><input type="text" value="abc"> (text)</p>
<p><input type="password" value="abc"> (password)</p>
<p><input type="checkbox" checked> (checkbox)</p>
<p><input type="radio" value="A" name="choice">
   <input type="radio" value="B" name="choice" checked>
   <input type="radio" value="C" name="choice"> (radio)</p>
<p><input type="file"> (file)</p>
```

这些元素的 JavaScript 接口和元素类型不同。

多行文本输入框有其自己的标签 `<textarea>`，这样做是因为通过一个属性来声明一个多行初始值会十分奇怪。`<textarea>` 要求有一个相匹配的 `</textarea>` 结束标签并使用标签之间的文本作为初始值，而不是使用 `value` 属性存储文本。

```
<textarea>
one
two
three
</textarea>
```

`<select>` 标签用来创造一个可以让用户从一些提前设定好的选项中进行选择的字段。

```
<select>
  <option>Pancakes</option>
  <option>Pudding</option>
  <option>Ice cream</option>
</select>
```

当一个表单字段中的内容更改时会触发 `change` 事件。

聚焦

不同于 HTML 文档中的其他元素，表单字段可以获取键盘焦点。当点击或以某种方式激活时，他们会成为激活的元素，并接受键盘的输入。

因此，只有获得焦点时，你才能输入文本字段。 其他字段对键盘事件的响应不同。 例如，`<select>` 菜单尝试移动到包含用户输入文本的选项，并通过向上和向下移动其选项来响应箭头键。

我们可以通过使用 JavaScript 的 `focus` 和 `blur` 方法来控制聚焦。第一个会聚焦到某一个 DOM 元素，第二个则使其失焦。在 `document.activeElement` 中的值会关联到当前聚焦的元素。

```
<input type="text">
<script>
  document.querySelector("input").focus();
  console.log(document.activeElement.tagName);
  // → INPUT
  document.querySelector("input").blur();
  console.log(document.activeElement.tagName);
  // → BODY
</script>
```

对于一些页面，用户希望立刻使用到一个表单字段。JavaScript 可以在页面载入完成时将焦点放到这些字段上，HTML 提供了 `autofocus` 属性，可以实现相同的效果，并让浏览器知道我们正在尝试实现的事情。这向浏览器提供了选项，来禁用一些错误的操作，例如用户希望将

焦点置于其他地方。

浏览器也允许用户通过 TAB 键来切换焦点。通过 `tabindex` 属性可以改变元素接受焦点的顺序。后面的例子会让焦点从文本输入框跳转到 OK 按钮而不是到帮助链接。

```
<input type="text" tabindex=1> <a href=".">(help)</a>
<button onclick="console.log('ok')" tabindex=2>OK</button>
```

默认情况下，多数的 HTML 元素不能拥有焦点。但是可以通过添加 `tabindex` 属性使任何元素可聚焦。 `tabindex` 为 -1 使 TAB 键跳过元素，即使它通常是可聚焦的。

禁用字段

所有的表单字段都可以通过其 `disable` 属性来禁用。它是一个可以被指定为没有值的属性 - 事实上它出现在所有禁用的元素中。

```
<button>I'm all right</button>
<button disabled>I'm out</button>
```

禁用的字段不能拥有焦点或更改，浏览器使它们变成灰色。

当一个程序在处理一些由按键或其他控制方式出发的事件，并且这些事件可能要求和服务器的通信时，将元素禁用直到动作完成可能是一个很好的方法。按照这用方式，当用户失去耐心并且再次点击时，不会意外的重复这一动作。

作为整体的表单

当一个字段被包含在 `<form>` 元素中时，其 DOM 元素会有一个 `form` 属性指向 `form` 的 DOM 元素。 `<form>` 元素则会有一个叫作 `elements` 属性，包含一个类似于数据的集合，其中包含全部的字段。

一个表单字段的 `name` 属性会决定在 `form` 提交时其内容的辨别方式。同时在获取 `form` 的 `elements` 属性时也可以作为一种属性名，所以 `elements` 属性既可以像数组（由编号来访问）一样使用也可以像映射一样访问（通过名字访问）。

```
<form action="example/submit.html">
  Name: <input type="text" name="name"><br>
  Password: <input type="password" name="password"><br>
  <button type="submit">Log in</button>
</form>
<script>
  let form = document.querySelector("form");
  console.log(form.elements[1].type);
  // → password
  console.log(form.elements.password.type);
  // → password
  console.log(form.elements.name.form == form);
  // → true
</script>
```

`type` 属性为 `submit` 的按钮在点击时，会提交表单。在一个 `form` 拥有焦点时，点击 `enter` 键也会有同样的效果。

通常在提交一个表单时，浏览器会将页面导航到 `form` 的 `action` 属性指明的页面，使用 `GET` 或 `POST` 请求。但是在这些发生之前， `"submit"` 事件会被触发。这个事件可以由 JavaScript 处理，并且处理器可以通过调用事件对象的 `preventDefault` 来禁用默认行为。

```
<form action="example/submit.html">
  Value: <input type="text" name="value">
  <button type="submit">Save</button>
</form>
<script>
  let form = document.querySelector("form");
  form.addEventListener("submit", event => {
    console.log("Saving value", form.elements.value.value);
    event.preventDefault();
  });
</script>
```

在 JavaScript 中 `submit` 事件有多种用途。我们可以编写代码来检测用户输入是否正确并且立刻提示错误信息，而不是提交表单。或者我们可以禁用正常的提交方式，正如这个例子中，让我们的程序处理输入，可能使用 `fetch` 将其发送到服务器而不重新加载页面。

文本字段

由 `type` 属性为 `text` 或 `password` 的 `<input>` 标签和 `textarea` 标签组成的字段有相同的接口。其 DOM 元素都有一个 `value` 属性，保存了为字符串格式的当前内容。将这个属性更改为另一个值将改变字段的内容。

文本字段 `selectionStart` 和 `selectEnd` 属性包含光标和所选文字的信息。当没有选中文字时，这两个属性的值相同，表明当前光标的信息。例如，0 表示文本的开始，10 表示光标在第十个字符之后。当一部分字段被选中时，这两个属性值会不同，表明选中文字开始位置和结束位置。

和正常的值一样，这些属性也可以被更改。

想象你正在编写关于 Knaseknemwy 的文章，但是名字拼写有一些问题，后续代码将`<textarea>`标签和一个事件处理器关联起来，当点击`F2`时，插入 Knaseknemwy。

```
<textarea></textarea>
<script>
  let textarea = document.querySelector("textarea");
  textarea.addEventListener("keydown", event => {
    // The key code for F2 happens to be 113
    if (event.keyCode == 113) {
      replaceSelection(textarea, "Khasekhemwy");
      event.preventDefault();
    }
  });
  function replaceSelection(field, word) {
    let from = field.selectionStart, to = field.selectionEnd;
    field.value = field.value.slice(0, from) + word +
                  field.value.slice(to);
    // Put the cursor after the word
    field.selectionStart = from + word.length;
    field.selectionEnd = from + word.length;
  }
</script>
```

`replaceSelection`函数用给定的字符串替换当前选中的文本字段内容，并将光标移动到替换内容后让用户可以继续输入。`change`事件不会在每次有输入时都被调用，而是在内容在改变并失焦后触发。为了及时的响应文本字段的改变，则需要为`input`事件注册一个处理器，每当用户有输入或更改时就被触发。

下面的例子展示一个文本字段和一个展示字段中的文字的当前长度的计数器。

```
<input type="text"> length: <span id="length">0</span>
<script>
  let text = document.querySelector("input");
  let output = document.querySelector("#length");
  text.addEventListener("input", () => {
    output.textContent = text.value.length;
  });
</script>
```

选择框和单选框

一个选择框只是一个双选切换。其值可以通过其包含一个布尔值的`checked`属性来获取和更改。

```
<label>
  <input type="checkbox" id="purple"> Make this page purple
</label>
<script>
  let checkbox = document.querySelector("#purple");
  checkbox.addEventListener("change", () => {
    document.body.style.background =
      checkbox.checked ? "mediumpurple" : "";
  });
</script>
```

`<label>` 标签关联部分文本和一个输入字段。点击标签上的任何位置将激活该字段，这样会将其聚焦，并当它为复选框或单选按钮时切换它的值。

单选框和选择框类似，不过单选框可以通过相同的 `name` 属性，隐式关联其他几个单选框，保证只能选择其中一个。

```
Color:
<label>
  <input type="radio" name="color" value="orange"> Orange
</label>
<label>
  <input type="radio" name="color" value="lightgreen"> Green
</label>
<label>
  <input type="radio" name="color" value="lightblue"> Blue
</label>
<script>
  let buttons = document.querySelectorAll("[name=color]");
  for (let button of Array.from(buttons)) {
    button.addEventListener("change", () => {
      document.body.style.background = button.value;
    });
  }
</script>
```

提供给 `querySelectorAll` 的 CSS 查询中的方括号用于匹配属性。 它选择 `name` 属性为 `"color"` 的元素。

选择字段

选择字段和单选按钮比较相似，允许用户从多个选项中选择。但是，单选框的展示排版是由我们控制的，而 `<select>` 标签外观则是由浏览器控制。

选择字段也有一个更类似于复选框列表的变体，而不是单选框。 当赋予 `multiple` 属性时， `<select>` 标签将允许用户选择任意数量的选项，而不仅仅是一个选项。 在大多数浏览器中，这会显示与正常的选择字段不同的效果，后者通常显示为下拉控件，仅在你打开它时才显示选项。

每一个 `<option>` 选项会有一个值，这个值可以通过 `value` 属性来定义。如果没有提供，选项内的文本将作为其值。 `<select>` 的 `value` 属性反映了当前的选中项。对于一个多选字段，这个属性用处不太大因为该属性只会给出一个选中项。

`<select>` 字段的 `<option>` 标签可以通过一个类似于数组对象的 `options` 属性访问到。每个选项会有一个叫作 `selected` 的属性，来表明这个选项当前是否被选中。这个属性可以用来被设定选中或不选中。

这个例子会从多选字段中取出选中的数值，并使用这些数值构造一个二进制数字。按住 `CTRL` （或 Mac 的 `COMMAND` 键）来选择多个选项。

```
<select multiple>
  <option value="1">0001</option>
  <option value="2">0010</option>
  <option value="4">0100</option>
  <option value="8">1000</option>
</select> = <span id="output">0</span>
<script>
  let select = document.querySelector("select");
  let output = document.querySelector("#output");
  select.addEventListener("change", () => {
    let number = 0;
    for (let option of Array.from(select.options)) {
      if (option.selected) {
        number += Number(option.value);
      }
    }
    output.textContent = number;
  });
</script>
```

文件字段

文件字段最初是用于通过表单来上传从浏览器机器中获取的文件。在现代浏览器中，也可以从 JavaScript 程序中读取文件。该字段则作为一个看门人角色。脚本不能简单地直接从用户的电脑中读取文件，但是如果用户在这个字段中选择了一个文件，浏览器会将这个行为解释为脚本，便可以访问该文件。

一个文本字段是一个类似于“选择文件”或“浏览”标签的按钮，后面跟着所选文件的信息。

```
<input type="file">
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    if (input.files.length > 0) {
      let file = input.files[0];
      console.log("You chose", file.name);
      if (file.type) console.log("It has type", file.type);
    }
  });
</script>
```

文本字段的 `files` 属性是一个类数组对象（当然，不是一个真正的数组），包含在字段中所选择的文件。开始时是空的。因此文本字段属性不仅仅是 `file` 属性。有时文本字段可以上传多个文件，这使得同时选择多个文件变为可能。

`files` 对象中的对象有 `name` （文件名）、 `size` （文件大小，单位为字节），和 `type` （文件的媒体类型，如 `text/plain` ， `image/jpeg` ）等属性。

而 `files` 属性中不包含文件内容的属性。获取这个内容会比较复杂。由于从硬盘中读取文件会需要一些时间，接口必须是异步的，来避免文档的无响应问题。

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("File", file.name, "starts with",
                    reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
  });
</script>
```

读取文件是通过`FileReader`对象实现的，注册一个`load`事件处理器，然后调用`readAsText`方法，传入我们希望读取的文件，一旦载入完成，`reader`的`result`属性内容就是文件内容。

`FileReader`对象还会在读取文件失败时触发`error`事件。错误对象本身会存在`reader`的`error`属性中。这个接口是在`Promise`成为语言的一部分之前设计的。你可以把它包装在`Promise`中，像这样：

```
function readFileText(file) {
  return new Promise((resolve, reject) => {
    let reader = new FileReader();
    reader.addEventListener(
      "load", () => resolve(reader.result));
    reader.addEventListener(
      "error", () => reject(reader.error));
    });
    reader.readAsText(file);
  });
}
```

客户端保存数据

采用 JavaScript 代码的简单 HTML 页面可以作为实现一些小应用的很好的途径。可以采用小的帮助程序来自动化一些基本的任务。通过关联一些表单字段和事件处理器，你可以实现华氏度与摄氏度的转换。也可以实现由主密码和网站名来生成密码等各种任务。

当一个应用需要存储一些东西以便于跨对话使用时，则不能使用 JavaScript 绑定因为每当页面关闭时这些值就会丢失。你可以搭建一个服务器，连接到因特网，将一些服务数据存储到其中。在第20章中将会介绍如何实现这些，当然这需要很多的工作，也有一定的复杂度。有时只要将数据存储在浏览器中即可。

`localStorage`对象可以用于保存数据，它在页面重新加载后还存在。这个对象允许你将字符串存储在某个名字（也是字符串）下，下面是具体示例。

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

一个在`localStorage`中的值会保留到其被重写时，它也可以通过`removeItem`来清除，或者由用户清除本地数据。

不同字段名的站点的数据会存在不同的地方。这也表明原则上由`localStorage`存储的数据只可以由相同站点的脚本编辑。

浏览器的确限制一个站点可以存储的`localStorage`的数据大小。这种限制，以及用垃圾填满人们的硬盘并不是真正有利可图的事实，防止该特性占用太多空间。

下面的代码实现了一个粗糙的笔记应用。程序将用户的笔记保存为一个对象，将笔记的标题和内容字符串相关联。对象被编码为 JSON 格式并存储在`localStorage`中。用户可以从`<select>`选择字段中选择笔记并在`<textarea>`中编辑笔记，并可以通过点击一个按钮来添加笔记。

```
Notes: <select></select> <button>Add</button><br>
<textarea style="width: 100%"></textarea>

<script>
  let list = document.querySelector("select");
  let note = document.querySelector("textarea");

  let state;
  function setState(newState) {
    list.textContent = "";
    for (let name of Object.keys(newState.notes)) {
      let option = document.createElement("option");
      option.textContent = name;
      if (newState.selected == name) option.selected = true;
      list.appendChild(option);
    }
    note.value = newState.notes[newState.selected];

    localStorage.setItem("Notes", JSON.stringify(newState));
    state = newState;
   }
  setState(JSON.parse(localStorage.getItem("Notes")) || {
    notes: {"shopping list": "Carrots\nRaisins"},
    selected: "shopping list"
  });
  }

  list.addEventListener("change", () => {
    setState({notes: state.notes, selected: list.value});
  });
  note.addEventListener("change", () => {
    setState({
      notes: Object.assign({}, state.notes,
                           {[state.selected]: note.value}),
      selected: state.selected
    });
  });

  document.querySelector("button")
    .addEventListener("click", () => {
      let name = prompt("Note name");
      if (name) setState({
        notes: Object.assign({}, state.notes, {[name]: ""}),
        selected: name
      });
    });
</script>
```

脚本从存储在`localStorage`中的`"Notes"`值来获取它的初始状态，如果其中没有值，它会创建示例状态，仅仅带有一个购物列表。从`localStorage`中读取不存在的字段会返回`null`。

`setState`方法确保 DOM 显示给定的状态，并将新状态存储到`localStorage`。 事件处理器调用这个函数来移动到一个新状态。

在这个例子中使用`Object.assign`，是为了创建一个新的对象，它是旧的`state.notes`的一个克隆，但是添加或覆盖了一个属性。 `Object.assign`选取第一个参数，向其添加所有更多参数的所有属性。 因此，向它提供一个空对象会使它填充一个新对象。 第三个参数中的方括号表示法，用于创建名称基于某个动态值的属性。

还有另一个和 `localStorage` 很相似的对象叫作 `sessionStorage` 。这两个对象之间的区别在于 `sessionStorage` 的内容会在每次会话结束时丢失，而对于多数浏览器来说，会话会在浏览器关闭时结束。

本章小结

在本章中，我们讨论了 HTTP 协议的工作原理。 客户端发送一个请求，该请求包含一个方法（通常是 `GET` ）和一个标识资源的路径。 然后服务器决定如何处理请求，并用状态码和响应正文进行响应。 请求和响应都可能包含提供附加信息的协议头。

浏览器 JavaScript 可以通过 `fetch` 接口生成 HTTP 请求。 像这样生成请求：

```
fetch("/18_http.html").then(r => r.text()).then(text => {
  console.log(`The page starts with ${text.slice(0, 15)}`);
});
```

浏览器生成 `GET` 请求来获取显示网页所需的资源。 页面也可能包含表单，这些表单允许在提交表单时，用户输入的信息发送为新页面的请求。

HTML可以表示多种表单字段，例如文本字段、选择框、多选字段和文件选取。

这些字段可以用 JavaScript 进行控制和读取。内容改变时会触发 `change` 事件，文本有输入时会触发 `input` 事件，键盘获得焦点时触发键盘事件。 例如 `"value"` （用于文本和选择字段）或 `"checked"` （用于复选框和单选按钮）的属性，用于读取或设置字段的内容。

当一个表单被提交时，会触发其 `submit` 事件，JavaScript 处理器可以通过调用 `preventDefault` 来禁用默认的提交事件。表单字段的元素不一定需要被包装在 `<form>` 标签中。

当用户在一个文件选择字段中选择了本机中的一个文件时，可以用 `FileReader` 接口来在 JavaScript 中获取文件内容。

`localStorage` 和 `sessionStorage` 对象可以用来保存页面重载后依旧保留的信息。第一个会永久保留数据（直到用户决定清除），第二个则会保存到浏览器关闭时。

习题

内容协商

HTTP 可以做的事情之一就是内容协商。 `Accept` 请求头用于告诉服务器，客户端想要获得什么类型的文档。 许多服务器忽略这个协议头，但是当一个服务器知道各种编码资源的方式时，它可以查看这个协议头，并发送客户端首选的格式。

URL `eloquentjavascript.net/author` 配置为响应明文，HTML 或 JSON，具体取决于客户端要求的内容。 这些格式由标准化的媒体类型 `"text/plain"` ， `"text/html"` 和 `"application/json"` 标识。

发送请求来获取此资源的所有三种格式。 使用传递给 `fetch` 的 `options` 对象中的 `headers` 属性，将名为 `Accept` 的协议头设置为所需的媒体类型。

最后，请尝试请求媒体类型 `"application/rainbows+unicorns"` ，并查看产生的状态码。

```
// Your code here.
```

JavaScript 工作台

构建一个接口，允许用户输入和运行一段 JavaScript 代码。

在 `<textarea>` 字段旁边放置一个按钮，当按下该按钮时，使用我们在第 10 章中看到的 `Function` 构造器，将文本包装到一个函数中并调用它。 将函数的返回值或其引发的任何错误转换为字符串，并将其显示在文本字段下。

```
<textarea id="code">return "hi";</textarea>
<button id="button">Run</button>
<pre id="output"></pre>

<script>
  // Your code here.
</script>
```

Conway 的生命游戏

Conway 的生命游戏是一个简单的在网格中模拟生命的游戏，每一个细胞都可以生存或灭亡。对于每一代（回合），都要遵循以下规则：

- 任何细胞，周围有少于两个或多于三个的活着的邻居，都会死亡。
- 任意细胞，拥有两个或三个的活着的邻居，可以生存到下一代。
- 任何死去的细胞，周围有三个活着的邻居，可以再次复活。

任意一个相连的细胞都可以称为邻居，包括对角相连。

注意这些规则要立刻应用于整个网格，而不是一次一个网格。这表明邻居的数目由开始的一代决定，并且邻居在每一代时发生的变化不应该影响给定细胞新的状态。

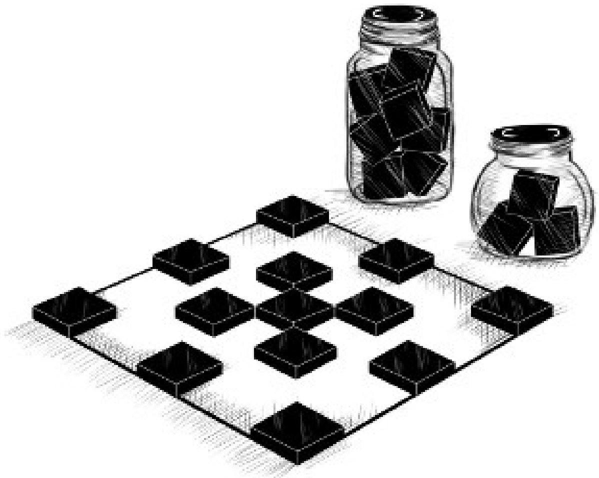
使用任何一个你认为合适的数据结构来实现这个游戏。使用 `Math.random` 来随机的生成开始状态。将其展示为一个选择框组成的网格和一个生成下一代的按钮。当用户选中或取消选中一个选择框时，其变化应该影响下一代的计算。

```
<div id="grid"></div>
<button id="next">Next generation</button>

<script>
  // Your code here.
</script>
```

十九、项目：像素艺术编辑器

> 原文：Project: A Pixel Art Editor
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译
>
> 我看着眼前的许多颜色。 我看着我的空白画布。 然后，我尝试使用颜色，就像形成诗歌的词语，就像塑造音乐的音符。
>
> Joan Miro



前面几章的内容为你提供了构建基本的 Web 应用所需的所有元素。 在本章中，我们将实现一个。

我们的应用将是像素绘图程序，你可以通过操纵放大视图（正方形彩色网格），来逐像素修改图像。 你可以使用它来打开图像文件，用鼠标或其他指针设备在它们上面涂画并保存。 这是它的样子：



在电脑上绘画很棒。 你不需要担心材料，技能或天赋。 你只需要开始涂画。

组件

应用的界面在顶部显示大的`<canvas>`元素，在它下面有许多表单字段。 用户通过从`<select>`字段中选择工具，然后单击，触摸或拖动画布来绘制图片。 有用于绘制单个像素或矩形，填充区域以及从图片中选取颜色的工具。

我们将编辑器界面构建为多个组件和对象，负责 DOM 的一部分，并可能在其中包含其他组件。

应用的状态由当前图片，所选工具和所选颜色组成。 我们将建立一些东西，以便状态存在于单一的值中，并且界面组件总是基于当前状态下他们看上去的样子。

为了明白为什么这很重要，让我们考虑替代方案：将状态片段分配给整个界面。 直到某个时期，这更容易编写。 我们可以放入颜色字段，并在需要知道当前颜色时读取其值。

但是，我们添加了颜色选择器。它是一种工具，可让你单击图片来选择给定像素的颜色。 为了保持颜色字段显示正确的颜色，该工具必须知道它存在，并在每次选择新颜色时对其进行更新。 如果你添加了另一个让颜色可见的地方（也许鼠标光标可以显示它），你必须更新你的改变颜色的代码来保持同步。

实际上，这会让你遇到一个问题，即界面的每个部分都需要知道所有其他部分，它们并不是非常模块化的。 对于本章中的小应用，这可能不成问题。 对于更大的项目，它可能变成真正的噩梦。

所以为了在原则上避免这种噩梦，我们将对数据流非常严格。 存在一个状态，界面根据该状态绘制。 界面组件可以通过更新状态来响应用户动作，此时组件有机会与新的状态进行同步。

在实践中，每个组件的建立，都是为了在给定一个新的状态时，它还会通知它的子组件，只要这些组件需要更新。 建立这个有点麻烦。 让这个更方便是许多浏览器编程库的主要卖点。但对于像这样的小应用，我们可以在没有这种基础设施的情况下完成。

状态更新表示为对象，我们将其称为动作。 组件可以创建这样的动作并分派它们 - 将它们给予中央状态管理函数。 该函数计算下一个状态，之后界面组件将自己更新为这个新状态。

我们正在执行一个混乱的任务，运行一个用户界面并对其应用一些结构。 尽管与 DOM 相关的部分仍然充满了副作用，但它们由一个概念上简单的主干支撑 - 状态更新循环。 状态决定了 DOM 的外观，而 DOM 事件可以改变状态的唯一方法，是向状态分派动作。

这种方法有许多变种，每个变种都有自己的好处和问题，但它们的中心思想是一样的：状态变化应该通过明确定义的渠道，而不是遍布整个地方。

我们的组件将是与界面一致的类。 他们的构造器被赋予一个状态，它可能是整个应用状态，或者如果它不需要访问所有东西，是一些较小的值，并使用它构建一个`dom`属性，也就是表示组件的 DOM。 大多数构造器还会接受一些其他值，这些值不会随着时间而改变，例如它们可用于分派操作的函数。

每个组件都有一个`setState`方法，用于将其同步到新的状态值。 该方法接受一个参数，该参数的类型与构造器的第一个参数的类型相同。

状态

应用状态将是一个带有图片，工具和颜色属性的对象。 图片本身就是一个对象，存储图片的宽度，高度和像素内容。 像素逐行存储在一个数组中，方式与第 6 章中的矩阵类相同，按行存储，从上到下。

```
class Picture {
  constructor(width, height, pixels) {
    this.width = width;
    this.height = height;
    this.pixels = pixels;
  }
  static empty(width, height, color) {
    let pixels = new Array(width * height).fill(color);
    return new Picture(width, height, pixels);
  }
  pixel(x, y) {
    return this.pixels[x + y * this.width];
  }
  draw(pixels) {
    let copy = this.pixels.slice();
    for (let {x, y, color} of pixels) {
      copy[x + y * this.width] = color;
    }
    return new Picture(this.width, this.height, copy);
  }
}
```

我们希望能够将图片当做不变的值，我们将在本章后面回顾其原因。 但是我们有时也需要一次更新大量像素。 为此，该类有 `draw` 方法，接受更新后的像素（具有 `x` ， `y` 和 `color` 属性的对象）的数组，并创建一个覆盖这些像素的新图像。 此方法使用不带参数的 `slice` 来复制整个像素数组 - 切片的起始位置默认为 0，结束位置为数组的长度。

`empty` 方法使用我们以前没有见过的两个数组功能。 可以使用数字调用 `Array` 构造器来创建给定长度的空数组。 然后 `fill` 方法可以用于使用给定值填充数组。 这些用于创建一个数组，所有像素具有相同颜色。

颜色存储为字符串，包含传统 CSS 颜色代码 - 一个井号（ `#` ），后跟六个十六进制数字，两个用于红色分量，两个用于绿色分量，两个用于蓝色分量。这是一种有点神秘而不方便的颜色编写方法，但它是 HTML 颜色输入字段使用的格式，并且可以在 `canva` s绘图上下文的 `fillColor` 属性中使用，所以对于我们在程序中使用颜色的方式，它足够实用。

所有分量都为零的黑色写成 `"#000000"` ，亮粉色看起来像 `#ff00ff"` ，其中红色和蓝色分量的最大值为 255，以十六进制数字写为 `ff` （ `a` 到 `f` 用作数字 10 到 15）。

我们将允许界面将动作分派为对象，它是属性覆盖先前状态的属性。当用户改变颜色字段时，颜色字段可以分派像 `{color: field.value}` 这样的对象，从这个对象可以计算出一个新的状态。

```
function updateState(state, action) {
  return Object.assign({}, state, action);
}
```

这是相当麻烦的模式，其中 `Object.assign` 用于首先将状态属性添加到空对象，然后使用来自动作的属性覆盖其中的一些属性，这在使用不可变对象的 JavaScript 代码中很常见。 一个更方便的表示法处于标准化的最后阶段，也就是在对象表达式中使用三点运算符来包含另一个对象的所有属性。 有了这个补充，你可以写出 `{...state, ...action}` 。 在撰写本文时，这还不适用于所有浏览器。

DOM 的构建

界面组件做的主要事情之一是创建 DOM 结构。 我们再也不想直接使用冗长的 DOM 方法，所以这里是 `elt` 函数的一个稍微扩展的版本。

```
function elt(type, props, ...children) {
  let dom = document.createElement(type);
  if (props) Object.assign(dom, props);
  for (let child of children) {
    if (typeof child != "string") dom.appendChild(child);
    else dom.appendChild(document.createTextNode(child));
  }
  return dom;
}
```

这个版本与我们在第 16 章中使用的版本之间的主要区别在于，它将属性（property）分配给 DOM 节点，而不是属性（attribute）。 这意味着我们不能用它来设置任意属性（attribute），但是我们可以用它来设置值不是字符串的属性（property），比如`onclick`，可以将它设置为一个函数，来注册点击事件处理器。

这允许这种注册事件处理器的方式：

```
<body>
  <script>
    document.body.appendChild(elt("button", {
      onclick: () => console.log("click")
    }, "The button"));
  </script>
</body>
```

画布

我们要定义的第一个组件是界面的一部分，它将图片显示为彩色框的网格。 该组件负责两件事：显示图片并将该图片上的指针事件传给应用的其余部分。

因此，我们可以将其定义为仅了解当前图片，而不是整个应用状态的组件。 因为它不知道整个应用是如何工作的，所以不能直接发送操作。 相反，当响应指针事件时，它会调用创建它的代码提供的回调函数，该函数将处理应用的特定部分。

```
const scale = 10;

class PictureCanvas {
  constructor(picture, pointerDown) {
    this.dom = elt("canvas", {
      onmousedown: event => this.mouse(event, pointerDown),
      ontouchstart: event => this.touch(event, pointerDown)
    });
    drawPicture(picture, this.dom, scale);
  }
  setState(picture) {
    if (this.picture == picture) return;
    this.picture = picture;
    drawPicture(this.picture, this.dom, scale);
  }
}
```

我们将每个像素绘制成一个`10x10`的正方形，由比例常数决定。 为了避免不必要的工作，该组件会跟踪其当前图片，并且仅当将`setState`赋予新图片时才会重绘。

实际的绘图功能根据比例和图片大小设置画布大小，并用一系列正方形填充它，每个像素一个。

```
function drawPicture(picture, canvas, scale) {
  canvas.width = picture.width * scale;
  canvas.height = picture.height * scale;
  let cx = canvas.getContext("2d");

  for (let y = 0; y < picture.height; y++) {
    for (let x = 0; x < picture.width; x++) {
      cx.fillStyle = picture.pixel(x, y);
      cx.fillRect(x * scale, y * scale, scale, scale);
    }
  }
}
```

当鼠标悬停在图片画布上，并且按下鼠标左键时，组件调用`pointerDown`回调函数，提供被点击图片坐标的像素位置。这将用于实现鼠标与图片的交互。回调函数可能会返回另一个回调函数，以便在按下按钮并且将指针移动到另一个像素时得到通知。

```
PictureCanvas.prototype.mouse = function(downEvent, onDown) {
  if (downEvent.button != 0) return;
  let pos = pointerPosition(downEvent, this.dom);
  let onMove = onDown(pos);
  if (!onMove) return;
  let move = moveEvent => {
    if (moveEvent.buttons == 0) {
      this.dom.removeEventListener("mousemove", move);
    } else {
      let newPos = pointerPosition(moveEvent, this.dom);
      if (newPos.x == pos.x && newPos.y == pos.y) return;
      pos = newPos;
      onMove(newPos);
    }
  };
  this.dom.addEventListener("mousemove", move);
};

function pointerPosition(pos, domNode) {
  let rect = domNode.getBoundingClientRect();
  return {x: Math.floor((pos.clientX - rect.left) / scale),
          y: Math.floor((pos.clientY - rect.top) / scale)};
}
```

由于我们知道像素的大小，我们可以使用`getBoundingClientRect`来查找画布在屏幕上的位置，所以可以将鼠标事件坐标（`clientX`和`clientY`）转换为图片坐标。它们总是向下取舍，以便它们指代特定的像素。

对于触摸事件，我们必须做类似的事情，但使用不同的事件，并确保我们在`"touchstart"`事件中调用`preventDefault`以防止滑动。

```
PictureCanvas.prototype.touch = function(startEvent,
                                          onDown) {
  let pos = pointerPosition(startEvent.touches[0], this.dom);
  let onMove = onDown(pos);
  startEvent.preventDefault();
  if (!onMove) return;
  let move = moveEvent => {
    let newPos = pointerPosition(moveEvent.touches[0],
                                 this.dom);
    if (newPos.x == pos.x && newPos.y == pos.y) return;
    pos = newPos;
    onMove(newPos);
  };
  let end = () => {
    this.dom.removeEventListener("touchmove", move);
    this.dom.removeEventListener("touchend", end);
  };
  this.dom.addEventListener("touchmove", move);
  this.dom.addEventListener("touchend", end);
};
```

对于触摸事件，`clientX` 和 `clientY` 不能直接在事件对象上使用，但我们可以在 `touches` 属性中使用第一个触摸对象的坐标。

应用

为了能够逐步构建应用，我们将主要组件实现为画布周围的外壳，以及一组动态工具和控件，我们将其传递给其构造器。

控件是出现在图片下方的界面元素。 它们为组件构造器的数组而提供。

工具是绘制像素或填充区域的东西。 该应用将一组可用工具显示为 `<select>` 字段。 当前选择的工具决定了，当用户使用指针设备与图片交互时，发生的事情。 它们作为一个对象而提供，该对象将出现在下拉字段中的名称，映射到实现这些工具的函数。 这个函数接受图片位置，当前应用状态和 `dispatch` 函数作为参数。 它们可能会返回一个移动处理器，当指针移动到另一个像素时，使用新位置和当前状态调用该函数。

```
class PixelEditor {
  constructor(state, config) {
    let {tools, controls, dispatch} = config;
    this.state = state;

    this.canvas = new PictureCanvas(state.picture, pos => {
      let tool = tools[this.state.tool];
      let onMove = tool(pos, this.state, dispatch);
      if (onMove) return pos => onMove(pos, this.state);
    });
    this.controls = controls.map(
      Control => new Control(state, config));
    this.dom = elt("div", {}, this.canvas.dom, elt("br"),
                   ...this.controls.reduce(
                     (a, c) => a.concat(" ", c.dom), []));
  }
  setState(state) {
    this.state = state;
    this.canvas.setState(state.picture);
    for (let ctrl of this.controls) ctrl.setState(state);
  }
}
```

指定给 `PictureCanvas` 的指针处理器，使用适当的参数调用当前选定的工具，如果返回了移动处理器，使其也接收状态。

所有控件在 `this.controls` 中构造并存储，以便在应用状态更改时更新它们。 `reduce` 的调用会在控件的 DOM 元素之间引入空格。 这样他们看起来并不那么密集。

第一个控件是工具选择菜单。 它创建 `<select>` 元素，每个工具带有一个选项，并设置 `"change"` 事件处理器，用于在用户选择不同的工具时更新应用状态。

```
class ToolSelect {
  constructor(state, {tools, dispatch}) {
    this.select = elt("select", {
      onchange: () => dispatch({tool: this.select.value})
    }, ...Object.keys(tools).map(name => elt("option", {
      selected: name == state.tool
    }, name)));
    this.dom = elt("label", null, " Tool: " , this.select);
  }
  setState(state) { this.select.value = state.tool; }
}
```

通过将标签文本和字段包装在 `<label>` 元素中，我们告诉浏览器该标签属于该字段，例如，你可以点击标签来聚焦该字段。

我们还需要能够改变颜色 - 所以让我们添加一个控件。 `type` 属性为颜色的 HTML `<input>` 元素为我们提供了专门用于选择颜色的表单字段。 这种字段的值始终是 `"#RRGGBB"` 格式（红色，绿色和蓝色分量，每种颜色两位数字）的 CSS 颜色代码。 当用户与它交互时，浏览器将显示一个颜色选择器界面。

该控件创建这样一个字段，并将其连接起来，与应用状态的 `color` 属性保持同步。

```
class ColorSelect {
  constructor(state, {dispatch}) {
    this.input = elt("input", {
      type: "color",
      value: state.color,
      onchange: () => dispatch({color: this.input.value})
    });
    this.dom = elt("label", null, " Color: " , this.input);
  }
  setState(state) { this.input.value = state.color; }
}
```

绘图工具

在我们绘制任何东西之前，我们需要实现一些工具，来控制画布上的鼠标或触摸事件的功能。

最基本的工具是绘图工具，它可以将你点击或轻触的任何像素，更改为当前选定的颜色。 它分派一个动作，将图片更新为一个版本，其中所指的像素赋为当前选定的颜色。

```
function draw(pos, state, dispatch) {
  function drawPixel({x, y}, state) {
    let drawn = {x, y, color: state.color};
    dispatch({picture: state.picture.draw([drawn])});
  }
  drawPixel(pos, state);
  return drawPixel;
}
```

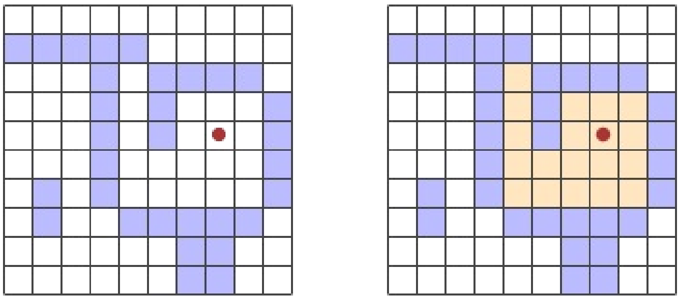
该函数立即调用`drawPixel`函数，但也会返回它，以便在用户在图片上拖动或滑动时，再次为新的所触摸的像素调用。

为了绘制较大的形状，可以快速创建矩形。 矩形工具在开始拖动的点和拖动到的点之间画一个矩形。

```
function rectangle(start, state, dispatch) {
  function drawRectangle(pos) {
    let xStart = Math.min(start.x, pos.x);
    let yStart = Math.min(start.y, pos.y);
    let xEnd = Math.max(start.x, pos.x);
    let yEnd = Math.max(start.y, pos.y);
    let drawn = [];
    for (let y = yStart; y <= yEnd; y++) {
      for (let x = xStart; x <= xEnd; x++) {
        drawn.push({x, y, color: state.color});
      }
    }
    dispatch({picture: state.picture.draw(drawn)});
  }
  drawRectangle(start);
  return drawRectangle;
}
```

此实现中的一个重要细节是，拖动时，矩形将从原始状态重新绘制在图片上。 这样，你可以在创建矩形时将矩形再次放大和缩小，中间的矩形不会在最终图片中残留。 这是不可变图片对象实用的原因之一 - 稍后我们会看到另一个原因。

实现洪水填充涉及更多东西。 这是一个工具，填充和指针下的像素，和颜色相同的所有相邻像素。“相邻”是指水平或垂直直接相邻，而不是对角线。 此图片表明，在标记像素处使用填充工具时，着色的一组像素：



有趣的是，我们的实现方式看起来有点像第 7 章中的寻路代码。那个代码搜索图来查找路线，但这个代码搜索网格来查找所有“连通”的像素。 跟踪一组可能的路线的问题是类似的。

```
const around = [{dx: -1, dy: 0}, {dx: 1, dy: 0},
                {dx: 0, dy: -1}, {dx: 0, dy: 1}];

function fill({x, y}, state, dispatch) {
  let targetColor = state.picture.pixel(x, y);
  let drawn = [{x, y, color: state.color}];
  for (let done = 0; done < drawn.length; done++) {
    for (let {dx, dy} of around) {
      let x = drawn[done].x + dx, y = drawn[done].y + dy;
      if (x >= 0 && x < state.picture.width &&
          y >= 0 && y < state.picture.height &&
          state.picture.pixel(x, y) == targetColor &&
          !drawn.some(p => p.x == x && p.y == y)) {
        drawn.push({x, y, color: state.color});
      }
    }
  }
  dispatch({picture: state.picture.draw(drawn)});
}
```

绘制完成的像素的数组可以兼作函数的工作列表。 对于每个到达的像素，我们必须看看任何相邻的像素是否颜色相同，并且尚未覆盖。 随着新像素的添加，循环计数器落后于绘制完成的数组的长度。 任何前面的像素仍然需要探索。 当它赶上长度时，没有剩下未探测的像素，并且该函数就完成了。

最终的工具是一个颜色选择器，它允许你指定图片中的颜色，来将其用作当前的绘图颜色。

```
function pick(pos, state, dispatch) {
  dispatch({color: state.picture.pixel(pos.x, pos.y)});
}
```

我们现在可以测试我们的应用了！

```
<div></div>
<script>
  let state = {
    tool: "draw",
    color: "#000000",
    picture: Picture.empty(60, 30, "#f0f0f0")
  };
  let app = new PixelEditor(state, {
    tools: {draw, fill, rectangle, pick},
    controls: [ToolSelect, ColorSelect],
    dispatch(action) {
      state = updateState(state, action);
      app.setState(state);
    }
  });
  document.querySelector("div").appendChild(app.dom);
</script>
```

保存和加载

当我们画出我们的杰作时，我们会想要保存它以备后用。 我们应该添加一个按钮，用于将当前图片下载为图片文件。 这个控件提供了这个按钮：

```
class SaveButton {
  constructor(state) {
    this.picture = state.picture;
    this.dom = elt("button", {
      onclick: () => this.save()
    }, "\u{1f4be} Save");
  }
  save() {
    let canvas = elt("canvas");
    drawPicture(this.picture, canvas, 1);
    let link = elt("a", {
      href: canvas.toDataURL(),
      download: "pixelart.png"
    });
    document.body.appendChild(link);
    link.click();
    link.remove();
  }
  setState(state) { this.picture = state.picture; }
}
```

组件会跟踪当前图片，以便在保存时可以访问它。 为了创建图像文件，它使用 `<canvas>` 元素来绘制图片（一比一的像素比例）。

`canvas` 元素上的 `toDataURL` 方法创建一个以 `data:` 开头的 URL。 与 `http:` 和 `https:` 的 URL 不同，数据 URL 在 URL 中包含整个资源。 它们通常很长，但它们允许我们在浏览器中，创建任意图片的可用链接。

为了让浏览器真正下载图片，我们将创建一个链接元素，指向此 URL 并具有 `download` 属性。 点击这些链接后，浏览器将显示一个文件保存对话框。 我们将该链接添加到文档，模拟点击它，然后再将其删除。

你可以使用浏览器技术做很多事情，但有时候做这件事的方式很奇怪。

并且情况变得更糟了。 我们也希望能够将现有的图像文件加载到我们的应用中。 为此，我们再次定义一个按钮组件。

```
class LoadButton {
  constructor(_, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => startLoad(dispatch)
    }, "\u{1f4c1} Load");
  }
  setState() {}
}

function startLoad(dispatch) {
  let input = elt("input", {
    type: "file",
    onchange: () => finishLoad(input.files[0], dispatch)
  });
  document.body.appendChild(input);
  input.click();
  input.remove();
}
```

为了访问用户计算机上的文件，我们需要用户通过文件输入字段选择文件。 但我不希望加载按钮看起来像文件输入字段，所以我们在单击按钮时创建文件输入，然后假装它自己被单击。

当用户选择一个文件时，我们可以使用`FileReader`访问其内容，并再次作为数据 URL。 该 URL 可用于创建`<img>`元素，但由于我们无法直接访问此类图像中的像素，因此我们无法从中创建`Picture`对象。

```
function finishLoad(file, dispatch) {
  if (file == null) return;
  let reader = new FileReader();
  reader.addEventListener("load", () => {
    let image = elt("img", {
      onload: () => dispatch({
        picture: pictureFromImage(image)
      }),
      src: reader.result
    });
  });
  reader.readAsDataURL(file);
}
```

为了访问像素，我们必须先将图片绘制到`<canvas>`元素。 `canvas`上下文有一个`getImageData`方法，允许脚本读取其像素。 所以一旦图片在画布上，我们就可以访问它并构建一个`Picture`对象。

```
function pictureFromImage(image) {
  let width = Math.min(100, image.width);
  let height = Math.min(100, image.height);
  let canvas = elt("canvas", {width, height});
  let cx = canvas.getContext("2d");
  cx.drawImage(image, 0, 0);
  let pixels = [];
  let {data} = cx.getImageData(0, 0, width, height);

  function hex(n) {
    return n.toString(16).padStart(2, "0");
  }
  for (let i = 0; i < data.length; i += 4) {
    let [r, g, b] = data.slice(i, i + 3);
    pixels.push("#" + hex(r) + hex(g) + hex(b));
  }
  return new Picture(width, height, pixels);
}
```

我们将图像的大小限制为`100×100`像素，因为任何更大的图像在我们的显示器上看起来都很大，并且可能会拖慢界面。

`getImageData`返回的对象的`data`属性，是一个颜色分量的数组。 对于由参数指定的矩形中的每个像素，它包含四个值，分别表示像素颜色的红色，绿色，蓝色和 alpha 分量，数字介于 0 和 255 之间。alpha 分量表示不透明度 - 当它是零时像素是完全透明的，当它是 255 时，它是完全不透明的。出于我们的目的，我们可以忽略它。

在我们的颜色符号中，为每个分量使用的两个十六进制数字，正好对应于 0 到 255 的范围 - 两个十六进制数字可以表示`16**2 = 256`个不同的数字。 数字的`toString`方法可以传入进制作为参数，所以`n.toString(16)`将产生十六进制的字符串表示。我们必须确保每个数字都占用两位数，所以十六进制的辅助函数调用`padStart`，在必要时添加前导零。

我们现在可以加载并保存了！ 在完成之前剩下一个功能。

撤销历史

编辑过程的一半是犯了小错误，并再次纠正它们。 因此，绘图程序中的一个非常重要的功能是撤消历史。

为了能够撤销更改，我们需要存储以前版本的图片。 由于这是一个不可变的值，这很容易。但它确实需要应用状态中的额外字段。

我们将添加`done`数组来保留图片的以前版本。 维护这个属性需要更复杂的状态更新函数，它将图片添加到数组中。

但我们不希望存储每一个更改，而是一定时间量之后的更改。 为此，我们需要第二个属性`doneAt`，跟踪我们上次在历史中存储图片的时间。

```
function historyUpdateState(state, action) {
  if (action.undo == true) {
    if (state.done.length == 0) return state;
    return Object.assign({}, state, {
      picture: state.done[0],
      done: state.done.slice(1),
      doneAt: 0
    });
  } else if (action.picture &&
             state.doneAt < Date.now() - 1000) {
    return Object.assign({}, state, action, {
      done: [state.picture, ...state.done],
      doneAt: Date.now()
    });
  } else {
    return Object.assign({}, state, action);
  }
}
```

当动作是撤消动作时，该函数将从历史中获取最近的图片，并生成当前图片。

或者，如果动作包含新图片，并且上次存储东西的时间超过了一秒（1000 毫秒），会更新 `done` 和 `doneAt` 属性来存储上一张图片。

撤消按钮组件不会做太多事情。 它在点击时分派撤消操作，并在没有任何可以撤销的东西时禁用自身。

```
class UndoButton {
  constructor(state, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => dispatch({undo: true}),
      disabled: state.done.length == 0
    }, " Undo" );
  }
  setState(state) {
    this.dom.disabled = state.done.length == 0;
  }
}
```

让我们绘图吧

为了建立应用，我们需要创建一个状态，一组工具，一组控件和一个分派函数。 我们可以将它们传递给 `PixelEditor` 构造器来创建主要组件。 由于我们需要在练习中创建多个编辑器，因此我们首先定义一些绑定。

```
const startState = {
  tool: "draw",
  color: "#000000",
  picture: Picture.empty(60, 30, "#f0f0f0"),
  done: [],
  doneAt: 0
};

const baseTools = {draw, fill, rectangle, pick};

const baseControls = [
  ToolSelect, ColorSelect, SaveButton, LoadButton, UndoButton
];

function startPixelEditor({state = startState,
                           tools = baseTools,
                           controls = baseControls}) {
  let app = new PixelEditor(state, {
    tools,
    controls,
    dispatch(action) {
      state = historyUpdateState(state, action);
      app.setState(state);
    }
  });
  return app.dom;
}
```

解构对象或数组时，可以在绑定名称后面使用`=`，来为绑定指定默认值，该属性在缺失或未定义时使用。`startPixelEditor`函数利用它来接受一个对象，包含许多可选属性作为参数。例如，如果你未提供`tools`属性，则`tools`将绑定到`baseTools`。

这就是我们在屏幕上获得实际的编辑器的方式：

```
<div></div>
<script>
  document.querySelector("div")
    .appendChild(startPixelEditor({}));
</script>
```

来吧，画一些东西。我会等着你。

为什么这个很困难

浏览器技术是惊人的。它提供了一组强大的界面积木，排版和操作方法，以及检查和调试应用的工具。你为浏览器编写的软件可以在几乎所有电脑和手机上运行。

与此同时，浏览器技术是荒谬的。你必须学习大量愚蠢的技巧和难懂的事实才能掌握它，而它提供的默认编程模型非常棘手，大多数程序员喜欢用几层抽象来封装它，而不是直接处理它。

虽然情况肯定有所改善，但它以增加更多元素来解决缺点的方式，改善了它 - 也创造了更多复杂性。数百万个网站使用的特性无法真正被取代。即使可能，也很难决定它应该由什么取代。

技术从不存在于真空中 - 我们受到我们的工具，以及产生它们的社会，经济和历史因素的制约。 这可能很烦人，但通常更加有效的是，试图理解现有的技术现实如何发挥作用，以及为什么它是这样 - 而不是对抗它，或者转向另一个现实。

新的抽象可能会有所帮助。 我在本章中使用的组件模型和数据流约定，是一种粗糙的抽象。 如前所述，有些库试图使用户界面编程更愉快。 在编写本文时，React 和 Angular 是主流选择，但是这样的框架带有整个全家桶。 如果你对编写 Web 应用感兴趣，我建议调查其中的一些内容，来了解它们的原理，以及它们提供的好处。

练习

我们的程序还有提升空间。让我们添加一些更多特性作为练习。

键盘绑定

将键盘快捷键添加到应用。 工具名称的第一个字母用于选择工具，
而`control-Z`或`command-Z`激活撤消工作。

通过修改`PixelEditor`组件来实现它。 为`<div>`元素包装添加`tabIndex`属性 0，以便它可以接收键盘焦点。 请注意，与`tabindex`属性对应的属性称为`tabIndex`，`I`大写，我们的`elt`函数需要属性名称。 直接在该元素上注册键盘事件处理器。 这意味着你必须先单击，触摸或按下 TAB 选择应用，然后才能使用键盘与其交互。

请记住，键盘事件具有`ctrlKey`和`metaKey`（用于 Mac 上的`Command`键）属性，你可以使用它们查看这些键是否被按下。

```
<div></div>
<script>
  // The original PixelEditor class. Extend the constructor.
  class PixelEditor {
    constructor(state, config) {
      let {tools, controls, dispatch} = config;
      this.state = state;

      this.canvas = new PictureCanvas(state.picture, pos => {
        let tool = tools[this.state.tool];
        let onMove = tool(pos, this.state, dispatch);
        if (onMove) {
          return pos => onMove(pos, this.state, dispatch);
        }
      });
      this.controls = controls.map(
        Control => new Control(state, config));
      this.dom = elt("div", {}, this.canvas.dom, elt("br"),
                     ...this.controls.reduce(
                       (a, c) => a.concat(" ", c.dom), []));
    }
    setState(state) {
      this.state = state;
      this.canvas.setState(state.picture);
      for (let ctrl of this.controls) ctrl.setState(state);
    }
  }

  document.querySelector("div")
    .appendChild(startPixelEditor({}));
</script>
```

高效绘图

绘图过程中，我们的应用所做的大部分工作都发生在`drawPicture`中。 创建一个新状态并更新 DOM 的其余部分的开销并不是很大，但重新绘制画布上的所有像素是相当大的工作量。

找到一种方法，通过重新绘制实际更改的像素，使`PictureCanvas`的`setState`方法更快。

请记住，`drawPicture`也由保存按钮使用，所以如果你更改它，请确保更改不会破坏旧用途，或者使用不同名称创建新版本。

另请注意，通过设置其`width`或`height`属性来更改`<canvas>`元素的大小，将清除它，使其再次完全透明。

```
<div></div>
<script>
  // Change this method
  PictureCanvas.prototype.setState = function(picture) {
    if (this.picture == picture) return;
    this.picture = picture;
    drawPicture(this.picture, this.dom, scale);
  };

  // You may want to use or change this as well
  function drawPicture(picture, canvas, scale) {
    canvas.width = picture.width * scale;
    canvas.height = picture.height * scale;
    let cx = canvas.getContext("2d");

    for (let y = 0; y < picture.height; y++) {
      for (let x = 0; x < picture.width; x++) {
        cx.fillStyle = picture.pixel(x, y);
        cx.fillRect(x * scale, y * scale, scale, scale);
      }
    }
  }

  document.querySelector("div")
    .appendChild(startPixelEditor({}));
</script>
```

圆

定义一个名为`circle`的工具，当你拖动时绘制一个实心圆。 圆的中心位于拖动或触摸手势开始的位置，其半径由拖动的距离决定。

```
<div></div>
<script>
  function circle(pos, state, dispatch) {
    // Your code here
  }

  let dom = startPixelEditor({
    tools: Object.assign({}, baseTools, {circle})
  });
  document.querySelector("div").appendChild(dom);
</script>
```

合适的直线

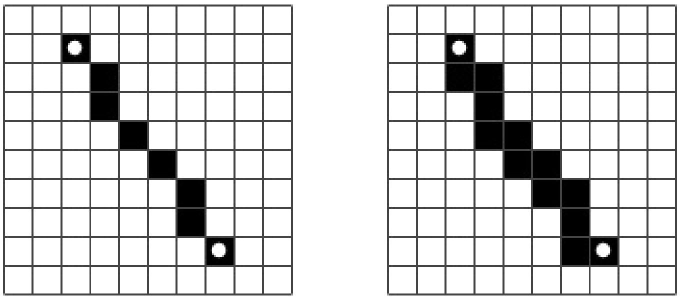
这是比前两个更高级的练习，它将要求你设计一个有意义的问题的解决方案。 在开始这个练习之前，确保你有充足的时间和耐心，并且不要因最初的失败而感到气馁。

在大多数浏览器上，当你选择绘图工具并快速在图片上拖动时，你不会得到一条闭合直线。相反，由于`"mousemove"`或`"touchmove"`事件没有快到足以命中每个像素，因此你会得到一些点，在它们之间有空隙。

改进绘制工具，使其绘制完整的直线。 这意味着你必须使移动处理器记住前一个位置，并将其连接到当前位置。

为此，由于像素可以是任意距离，所以你必须编写一个通用的直线绘制函数。

两个像素之间的直线是连接像素的链条，从起点到终点尽可能直。对角线相邻的像素也算作连接。 所以斜线应该看起来像左边的图片，而不是右边的图片。



如果我们有了代码，它在两个任意点间绘制一条直线，我们不妨继续，并使用它来定义 `line` 工具，它在拖动的起点和终点之间绘制一条直线。

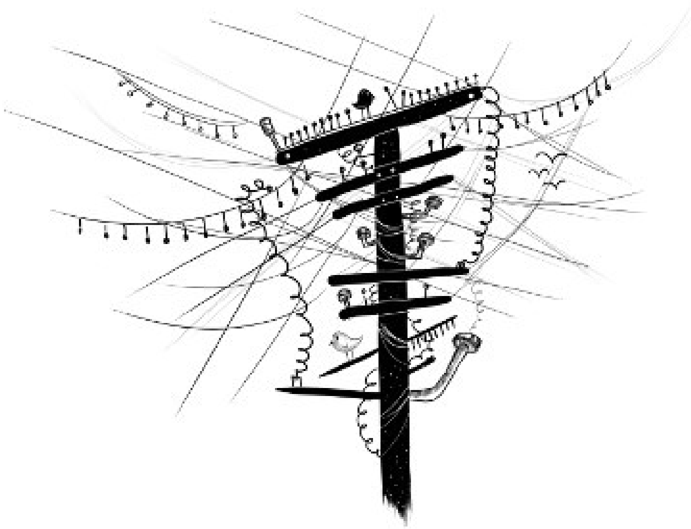
```
<div></div>
<script>
  // The old draw tool. Rewrite this.
  function draw(pos, state, dispatch) {
    function drawPixel({x, y}, state) {
      let drawn = {x, y, color: state.color};
      dispatch({picture: state.picture.draw([drawn])});
    }
    drawPixel(pos, state);
    return drawPixel;
  }

  function line(pos, state, dispatch) {
    // Your code here
  }

  let dom = startPixelEditor({
    tools: {draw, line, fill, rectangle, pick}
  });
  document.querySelector("div").appendChild(dom);
</script>
```

二十、Node.js

> 原文：Node.js
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> A student asked 'The programmers of old used only simple machines and no programming languages, yet they made beautiful programs. Why do we use complicated machines and programming languages?'. Fu-Tzu replied 'The builders of old used only sticks and clay, yet they made beautiful huts.'
>
> Master Yuan-Ma，《The Book of Programming》



到目前为止，我们已经使用了 JavaScript 语言，并将其运用于单一的浏览器环境中。本章和下一章将会大致介绍 Node.js，该程序可以让读者将你的 JavaScirpt 技能运用于浏览器之外。读者可以运用 Node.js 构建应用程序，实现简单的命令行工具和复杂动态 HTTP 服务器。

这些章节旨在告诉你建立 Node.js 的主要概念，并向你提供信息，使你可以采用 Nodejs 编写一些实用程序。它们并不是这个平台的完整的介绍。

如果你想要运行本章中的代码，需要安装 Node.js 10 或更高版本。 为此，请访问 nodejs.org，并按照用于你的操作系统的安装说明进行操作。 你也可以在那里找到 Node.js 的更多文档。

背景

编写通过网络通信的系统时，一个更困难的问题是管理输入输出，即向/从网络和硬盘读写数据。到处移动数据会耗费时间，而调度这些任务的技巧会使得系统在相应用户或网络请求时产生巨大的性能差异。

在这样的程序中，异步编程通常是有帮助的。 它允许程序同时向/从多个设备发送和接收数据，而无需复杂的线程管理和同步。

Node最初是为了使异步编程简单方便而设计的。 JavaScript 很好地适应了像 Node 这样的系统。 它是少数几种没有内置输入和输出方式的编程语言之一。 因此，JavaScript 可以适应 Node 的相当古怪的输入和输出方法，而不会产生两个不一致的接口。 在 2009 年设计 Node 时，人们已经在浏览器中进行基于回调的编程，所以该语言的社区用于异步编程风格。

Node 命令

在系统中安装完 Node.js 后，Node.js 会提供一个名为 `node` 的程序，该程序用于执行 JavaScript 文件。假设你有一个文件 hello.js，该文件会包含以下代码。

```
let message = "Hello world";
console.log(message);
```

读者可以仿照下面这种方式通过命令行执行程序。

```
$ node hello.js
Hello world
```

Node 中的 `console.log` 方法与浏览器中所做的类似，都用于打印文本片段。但在 Node 中，该方法不会将文本显示在浏览器的 JavaScript 控制台中，而显示在标准输出流中。从命令行运行 `node` 时，这意味着你会在终端中看到记录的值。

若你执行 `node` 时不附带任何参数， `node` 会给出提示符，读者可以输入 JavaScript 代码并立即看到执行结果。

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

`process` 绑定类似于 `console` 绑定，是 Node 中的全局绑定。该绑定提供了多种方式来监听并操作当前程序。该绑定中的 `exit` 方法可以结束进程并赋予一个退出状态码，告知启动 `node` 的程序（在本例中时命令行 Shell），当前程序是成功完成（代码为 0），还是遇到

了错误（其他代码）。

读者可以读取`process.argv`来获取传递给脚本的命令行参数，该绑定是一个字符串数组。请注意该数组包括了`node`命令和脚本名称，因此实际的参数从索引 2 处开始。若`showargv.js`只包含一条`console.log(process.argv)`语句，你可以这样执行该脚本。

```
$ node showargv.js one --and two
["node", "/tmp/showargv.js", "one", "--and", "two"]
```

所有标准 JavaScript 全局绑定，比如`Array`、`Math`以及`JSON`也都存在于 Node 环境中。而与浏览器相关的功能，比如`document`与`alert`则不存在。

模块

除了前文提到的一些绑定，比如`console`和`process`，Node 在全局作用域中添加了很少绑定。如果你需要访问其他的内建功能，可以通过`system`模块获取。

第十章中描述了基于`require`函数的 CommonJS 模块系统。该系统是 Node 的内建模块，用于在程序中装载任何东西，从内建模块，到下载的包，再到普通文件都可以。

调用`require`时，Node 会将给定的字符串解析为可加载的实际文件。路径名若以`"/"`、`"./"`或`"../"`开头，则解析为相对于当前模块的路径，其中`"./"`表示当前路径，`"../"`表示当前路径的上一级路径，而`"/"`则表示文件系统根路径。因此若你访问从文件`/tmp/robot/robot.js`访问`"./graph"`，Node 会尝试加载文件`/tmp/robot/graph.js`。

`.js`扩展名可能会被忽略，如果这样的文件存在，Node 会添加它。 如果所需的路径指向一个目录，则 Node 将尝试加载该目录中名为`index.js`的文件。

当一个看起来不像是相对路径或绝对路径的字符串被赋给`require`时，按照假设，它引用了内置模块，或者安装在`node_modules`目录中模块。 例如，`require("fs")`会向你提供 Node 内置的文件系统模块。 而`require("robot")`可能会尝试加载`node_modules/robot/`中的库。 安装这种库的一种常见方法是使用 NPM，我们稍后讲讲它。

我们来建立由两个文件组成的小项目。 第一个称为`main.js`，并定义了一个脚本，可以从命令行调用来反转字符串。

```
const {reverse} = require("./reverse");

// Index 2 holds the first actual command-line argument
let argument = process.argv[2];

console.log(reverse(argument));
```

文件`reverse.js`中定义了一个库，用于截取字符串，这个命令行工具，以及其他需要直接访问字符串反转函数的脚本，都可以调用该库。

```
exports.reverse = function(string) {
  return Array.from(string).reverse().join("");
};
```

请记住，将属性添加到 `exports` ，会将它们添加到模块的接口。 由于 Node.js 将文件视为 CommonJS 模块，因此 `main.js` 可以从 `reverse.js` 获取导出的 `reverse` 函数。

我们可以看到我们的工具执行结果如下所示。

```
$ node main.js JavaScript
tpircSavaJ
```

使用 NPM 安装

第十章中介绍的 NPM，是一个 JavaScript 模块的在线仓库，其中大部分模块是专门为 Node 编写的。当你在计算机上安装 Node 时，你就会获得一个名为 `npm` 的程序，提供了访问该仓库的简易界面。

它的主要用途是下载包。 我们在第十章中看到了 `ini` 包。 我们可以使用 NPM 在我们的计算机上获取并安装该包。

```
$ npm install ini
npm WARN enoent ENOENT: no such file or directory,
         open '/tmp/package.json'
+ ini@1.3.5
added 1 package in 0.552s
$ node
> const {parse} = require("ini");
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

运行 `npm install` 后，NPM 将创建一个名为 `node_modules` 的目录。 该目录内有一个包含库的 `ini` 目录。 你可以打开它并查看代码。 当我们调用 `require("ini")` 时，加载这个库，我们可以调用它的 `parse` 属性来解析配置文件。

默认情况下，NPM 在当前目录下安装包，而不是在中央位置。 如果你习惯于其他包管理器，这可能看起来很不寻常，但它具有优势 - 它使每个应用程序完全控制它所安装的包，并且使其在删除应用程序时，更易于管理版本和清理。

包文件

在 `npm install` 例子中，你可以看到 `package.json` 文件不存在的警告。 建议为每个项目创建一个文件，手动或通过运行 `npm init` 。 它包含该项目的一些信息，例如其名称和版本，并列出其依赖项。

来自第七章的机器人模拟，在第十章中模块化，它可能有一个`package.json`文件，如下所示：

```
{
  "author": "Marijn Haverbeke",
  "name": "eloquent-javascript-robot",
  "description": "Simulation of a package-delivery robot",
  "version": "1.0.0",
  "main": "run.js",
  "dependencies": {
    "dijkstrajs": "^1.0.1",
    "random-item": "^1.0.0"
  },
  "license": "ISC"
}
```

当你运行`npm install`而没有指定安装包时，NPM 将安装`package.json`中列出的依赖项。 当你安装一个没有列为依赖项的特定包时，NPM会将它添加到`package.json`中。

版本

`package.json`文件列出了程序自己的版本和它的依赖的版本。 版本是一种方式，用于处理包的单独演变。为使用某个时候的包而编写的代码，可能不能使用包的更高版本。

NPM 要求其包遵循名为语义版本控制（semantic versioning）的纲要，它编码了版本号中的哪些版本是兼容的（不破坏就接口）。 语义版本由三个数字组成，用点分隔，例如`2.3.0`。 每次添加新功能时，中间数字都必须递增。 每当破坏兼容性时，使用该包的现有代码可能不适用于新版本，因此必须增加第一个数字。

`package.json`中的依赖项版本号前面的脱字符（`^`），表示可以安装兼容给定编号的任何版本。 例如`"^2.3.0"`意味着任何大于等于`2.3.0`且小于`3.0.0`的版本都是允许的。

`npm`命令也用于发布新的包或包的新版本。 如果你在一个包含`package.json`文件的目录中执行`npm publish`，它将一个包发布到注册处，带有 JSON 文件中列出的名称和版本。 任何人都可以将包发布到 NPM - 但只能用新名称，因为任何人可以更新现有的包，会有点恐怖。

由于`npm`程序是与开放系统（包注册处）进行对话的软件，因此它没有什么独特之处。 另一个程序`yarn`，可以从 NPM 注册处中安装，使用一种不同的接口和安装策略，与`npm`具有相同的作用。

本书不会深入探讨 NPM 的使用细节。 请参阅`npmjs.org`来获取更多文档和搜索包的方法。

文件系统模块

在Node中最常用的内建模块就是`fs`（表示 filesystem，文件系统）模块。该模块提供了处理文件和目录的函数。

例如，有个函数名为 `readFile`，该函数读取文件并调用回调，并将文件内容传递给回调。

```
let {readFile} = require("fs");
readFile("file.txt", "utf8", (error, text) => {
  if (error) throw error;
  console.log("The file contains:", text);
});
```

`readFile` 的第二个参数表示字符编码，用于将文件解码成字符串。将文本编码成二进制数据有许多方式，但大多数现代系统使用 UTF-8，因此除非有特殊原因确信文件使用了别的编码，否则读取文件时使用 `"utf-8"` 是一种较为安全的方式。若你不传递任何编码，Node 会认为你需要解析二进制数据，因此会返回一个 `Buffer` 对象而非字符串。该对象类似于数组，每个元素是文件中字节（8 位的数据块）对应的数字。

```
const {readFile} = require("fs");
readFile("file.txt", (error, buffer) => {
  if (error) throw error;
  console.log("The file contained", buffer.length, "bytes.",
              "The first byte is:", buffer[0]);
});
```

有一个名为 `writeFile` 的函数与其类似，用于将文件写到磁盘上。

```
const {writeFile} = require("fs");
writeFile("graffiti.txt", "Node was here", err => {
  if (err) console.log(`Failed to write file: ${err}`);
  else console.log("File written.");
});
```

这里我们不需要制定编码，因为如果我们调用 `writeFile` 时传递的是字符串而非 `Buffer` 对象，则 `writeFile` 会使用默认编码（即 UTF-8）来输出文本。

`fs` 模块也包含了其他实用函数，其中 `readdir` 函数用于将目录中的文件以字符串数组的方式返回，`stat` 函数用于获取文件信息，`rename` 函数用于重命名文件，`unlink` 用于删除文件等。

而且其中大多数都将回调作为最后一个参数，它们会以错误（第一个参数）或成功结果（第二个参数）来调用。 我们在第十一章中看到，这种编程风格存在缺点 - 最大的缺点是，错误处理变得冗长且容易出错。

相关细节请参见http://nodejs.org/中的文档。

虽然 `Promise` 已经成为 JavaScript 的一部分，但是，将它们与 Node.js 的集成的工作仍然还在进行中。 从 v10 开始，标准库中有一个名为 `fs/promises` 的包，它导出的函数与 `fs` 大部分相同，但使用 `Promise` 而不是回调。

```
const {readFile} = require("fs/promises");
readFile("file.txt", "utf8")
  .then(text => console.log("The file contains:", text));
```

有时候你不需要异步，而是需要阻塞。`fs` 中的许多函数也有同步的变体，它们的名称相同，末尾加上 `Sync`。例如，`readFile` 的同步版本称为 `readFileSync`。

```
const {readFileSync} = require("fs");
console.log("The file contains:",
            readFileSync("file.txt", "utf8"));
```

请注意，在执行这样的同步操作时，程序完全停止。如果它应该响应用户或网络中的其他计算机，那么可在同步操作中可能会产生令人讨厌的延迟。

HTTP 模块

另一个主要模块名为 `"http"`。该模块提供了执行 HTTP 服务和产生 HTTP 请求的函数。

启动一个 HTTP 服务器只需要以下代码。

```
const {createServer} = require("http");
let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Hello!</h1>
    <p>You asked for <code>${request.url}</code></p>`);
  response.end();
});
server.listen(8000);
```

若你在自己的机器上执行该脚本，你可以打开网页浏览器，并访问 http://localhost:8000/hello，就会向你的服务器发出一个请求。服务器会响应一个简单的 HTML 页面。

每次客户端尝试连接服务器时，服务器都会调用传递给 `createServer` 函数的参数。`request` 和 `response` 绑定都是对象，分别表示输入数据和输出数据。`request` 包含请求信息，例如该对象的 `url` 属性表示请求的 URL。

因此，当你在浏览器中打开该页面时，它会向你自己的计算机发送请求。这会导致服务器函数运行并返回一个响应，你可以在浏览器中看到该响应。

你需要调用 `response` 对象的方法以将一些数据发回客户端。第一个函数调用（`writeHead`）会输出响应头（参见第十七章）。你需要向该函数传递状态码（本例中 200 表示成功）和一个对象，该对象包含协议头信息的值。该示例设置了 `"Content-Type"` 头，通知客户端我们将发送一个 HTML 文档。

接下来使用 `response.write` 来发送响应体（文档自身）。若你想一段一段地发送相应信息，可以多次调用该方法，例如将数据发送到客户端。最后调用 `response.end` 发送相应结束信号。

调用 `server.listen` 会使服务器在 8000 端口上开始等待请求。这就是你需要连接 `localhost:8000` 和服务器通信，而不是 `localhost`（这样将会使用默认端口，即 80）的原因。

当你运行这个脚本时，这个进程就在那里等着。 当一个脚本正在监听事件时 - 这里是网络连接 - Node 不会在到达脚本末尾时自动退出。为了关闭它，请按 `Ctrl-C`。

一个真实的 Web 服务器需要做的事情比示例多得多。其差别在于我们需要根据请求的方法（`method` 属性），来判断客户端尝试执行的动作，并根据请求的 URL 来找出动作处理的资源。本章随后会介绍更高级的服务器。

我们可以使用 `http` 模块的 `request` 函数来充当一个 HTTP 客户端。

```
const {request} = require("http");
let requestStream = request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, response => {
  console.log("Server responded with status code",
              response.statusCode);
});
requestStream.end();
```

`request` 函数的第一个参数是请求配置，告知 Node 需要访问的服务器、服务器请求地址、使用的方法等信息。第二个参数是响应开始时的回调。该回调会接受一个参数，用于检查相应信息，例如获取状态码。

和在服务器中看到的 `response` 对象一样，`request` 返回的对象允许我们使用 `write` 方法多次发送数据，并使用 `end` 方法结束发送。本例中并没有使用 `write` 方法，因为 GET 请求的请求正文中无法包含数据。

`https` 模块中有类似的 `request` 函数，可以用来向 `https:` URL 发送请求。

但是使用 Node 的原始功能发送请求相当麻烦。 NPM 上有更多方便的包装包。 例如，`node-fetch` 提供了我们从浏览器得知的，基于 `Promise` 的 `fetch` 接口。

流

我们在 HTTP 中看过两个可写流的例子，即服务器可以向 `response` 对象中写入数据，而 `request` 返回的请求对象也可以写入数据。

可写流是 Node 中广泛使用的概念。这种对象拥有 `write` 方法，你可以传递字符串或 `Buffer` 对象，来向流写入一些数据。它们 `end` 方法用于关闭流，并且还可以接受一个可选值，在流关闭之前将其写入流。 这两个方法也可以接受回调作为附加参数，当写入或关闭完成时它们将被调用。

我们也可以使用 `fs` 模块的 `createWriteStream`，建立一个指向本地文件的输出流。你可以调用该方法返回的结果对象的 `write` 方法，每次向文件中写入一段数据，而不是像 `writeFile` 那样一次性写入所有数据。

可读流则略为复杂。传递给 HTTP 服务器回调的 `request` 绑定，以及传递给 HTTP 客户端回调的 `response` 对象都是可读流（服务器读取请求并写入响应，而客户端则先写入请求，然后读取响应）。读取流需要使用事件处理器，而不是方法。

Node 中发出的事件都有一个 `on` 方法，类似浏览器中的 `addEventListener` 方法。该方法接受一个事件名和一个函数，并将函数注册到事件上，接下来每当指定事件发生时，都会调用注册的函数。

可读流有 `data` 事件和 `end` 事件。`data` 事件在每次数据到来时触发，`end` 事件在流结束时触发。该模型适用于“流”数据，这类数据可以立即处理，即使整个文档的数据没有到位。我们可以使用 `createReadStream` 函数创建一个可读流，来读取本地文件。

这段代码创建了一个服务器并读取请求正文，然后将读取到的数据全部转换成大写，并使用流写回客户端。

```
const {createServer} = require("http");
createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()));
  request.on("end", () => response.end());
  });
}).listen(8000);
```

传递给 `data` 处理器的 `chunk` 值是一个二进制 `Buffer` 对象，我们可以使用它的 `toString` 方法，通过将其解码为 UTF-8 编码的字符，来将其转换为字符串。

下面的一段代码，和上面的服务（将字母转换成大写）一起运行时，它会向服务器发送一个请求并输出获取到的响应数据：

```
const {request} = require("http");
request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, response => {
  response.on("data", chunk =>
    process.stdout.write(chunk.toString()));
}).end("Hello server");
// → HELLO SERVER
```

该示例代码向 `process.stdout`（进程的标准输出流，是一个可写流）中写入数据，而不使用 `console.log`，因为 `console.log` 函数会在输出的每段文本后加上额外的换行符，在这里不太合适。

文件服务器

让我们结合新学习的 HTTP 服务器和文件系统的知识，并建立起两者之间的桥梁：使用 HTTP 服务允许客户远程访问文件系统。这个服务有许多用处，它允许网络应用程序存储并共享数据或使得一组人可以共享访问一批文件。

当我们将文件当作 HTTP 资源时，可以将 HTTP 的 GET、PUT 和 DELETE 方法分别看成读取、写入和删除文件。我们将请求中的路径解释成请求指向的文件路径。

我们可能不希望共享整个文件系统，因此我们将这些路径解释成以服务器工作路径（即启动服务器的路径）为起点的相对路径。若从 `/home/marijn/public` （或 Windows 下的 `C:\Users\marijn\public` ）启动服务器，那么对 `/file.txt` 的请求应该指向 `/home/marijn/public/file.txt` （或 `C:\Users\marijn\public\file.txt` ）。

我们将一段段地构建程序，使用名为 `methods` 的对象来存储处理多种 HTTP 方法的函数。方法处理器是 `async` 函数，它接受请求对象作为参数并返回一个 `Promise` ，解析为描述响应的对象。

```
const {createServer} = require("http");

const methods = Object.create(null);

createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(error), status: 500};
    })
    .then(({body, status = 200, type = "text/plain"}) => {
       response.writeHead(status, {"Content-Type": type});
       if (body && body.pipe) body.pipe(response);
       else response.end(body);
    });
}).listen(8000);

async function notAllowed(request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  };
}
```

这样启动服务器之后，服务器永远只会产生 405 错误响应，该代码表示服务器拒绝处理特定的方法。

当请求处理程序的 `Promise` 受到拒绝时， `catch` 调用会将错误转换为响应对象（如果它还不是），以便服务器可以发回错误响应，来通知客户端它未能处理请求。

响应描述的 `status` 字段可以省略，这种情况下，默认为 200（OK）。 `type` 属性中的内容类型也可以被省略，这种情况下，假定响应为纯文本。

当 `body` 的值是可读流时，它将有 `pipe` 方法，用于将所有内容从可读流转发到可写流。 如果不是，则假定它是 `null` （无正文），字符串或缓冲区，并直接传递给响应的 `end` 方法。

为了弄清哪个文件路径对应于请求URL， `urlPath` 函数使用 Node 的 `url` 内置模块来解析 URL。 它接受路径名，类似 `"/file.txt"` ，将其解码来去掉 `%20` 风格的转义代码，并相对于程序的工作目录来解析它。

```
const {parse} = require("url");
const {resolve} = require("path");

const baseDirectory = process.cwd();

function urlPath(url) {
  let {pathname} = parse(url);
  let path = resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + "/")) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

只要你建立了一个接受网络请求的程序，就必须开始关注安全问题。 在这种情况下，如果我们不小心，很可能会意外地将整个文件系统暴露给网络。

文件路径在 Node 中是字符串。 为了将这样的字符串映射为实际的文件，需要大量有意义的解释。 例如，路径可能包含 `"../"` 来引用父目录。 因此，一个显而易见的问题来源是像 `/../ secret_file` 这样的路径请求。

为了避免这种问题， `urlPath` 使用 `path` 模块中的 `resolve` 函数来解析相对路径。 然后验证结果位于工作目录下面。 `process.cwd` 函数（其中 `cwd` 代表“当前工作目录”）可用于查找此工作目录。 当路径不起始于基本目录时，该函数将使用 HTTP 状态码来抛出错误响应对象，该状态码表明禁止访问资源。

我们需要创建GET方法，在读取目录时返回文件列表，在读取普通文件时返回文件内容。

一个棘手的问题是我们返回文件内容时添加的 `Content-Type` 头应该是什么类型。因为这些文件可以是任何内容，我们的服务器无法简单地对所有文件返回相同的内容类型。但 NPM 可以帮助我们完成该任务。 `mime` 包（以 `text/plain` 这种方式表示的内容类型，名为 MIME 类型）可以获取大量文件扩展名的正确类型。

以下 `npm` 命令在服务器脚本所在的目录中，安装 `mime` 的特定版本。

```
$ npm install mime@2.2.0
```

当请求文件不存在时，应该返回的正确 HTTP 状态码是 404。我们使用 `stat` 函数，来找出特定文件是否存在以及是否是一个目录。

```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs/promises");
const mime = require("mime");

methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: mime.getType(path)};
  }
};
```

因为 `stat` 访问磁盘需要耗费一些时间，因此该函数是异步的。由于我们使用 `Promise` 而不是回调风格，因此必须从 `fs/promises` 而不是 `fs` 导入。

当文件不存在时，`stat` 会抛出一个错误对象，`code` 属性为 `'ENOENT'`。这些有些模糊的，受 Unix 启发的代码，是你识别 Node 中的错误类型的方式。

由 `stat` 返回的 `stats` 对象告诉了我们文件的一系列信息，比如文件大小（`size` 属性）和修改日期（`mtime` 属性）。这里我们想知道的是，该文件是一个目录还是普通文件，`isDirectory` 方法可以告诉我们答案。

我们使用 `readdir` 来读取目录中的文件列表，并将其返回给客户端。对于普通文件，我们使用 `createReadStream` 创建一个可读流，并将其传递给 `respond` 对象，同时使用 `mime` 模块根据文件名获取内容类型并传递给 `respond`。

处理 `DELETE` 请求的代码就稍显简单了。

```
const {rmdir, unlink} = require("fs/promises");

methods.DELETE = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 204};
  }
  if (stats.isDirectory()) await rmdir(path);
  else await unlink(path);
  return {status: 204};
};
```

当 HTTP 响应不包含任何数据时，状态码 204（“No Content”，无内容）可用于表明这一点。由于删除的响应不需要传输任何信息，除了操作是否成功之外，在这里返回是明智的。

你可能想知道，为什么试图删除不存在的文件会返回成功状态代码，而不是错误。当被删除的文件不存在时，可以说该请求的目标已经完成。HTTP 标准鼓励我们使请求是幂等（idempotent）的，这意味着，多次发送相同请求的结果，会与一次相同。从某种意义上说，如果你试图删除已经消失的东西，那么你试图去做的效果已经实现 - 东西已经不存在了。

下面是 `PUT` 请求的处理器。

```
const {createWriteStream} = require("fs");

function pipeStream(from, to) {
  return new Promise((resolve, reject) => {
    from.on("error", reject);
    to.on("error", reject);
    to.on("finish", resolve);
    from.pipe(to);
  });
}

methods.PUT = async function(request) {
  let path = urlPath(request.url);
  await pipeStream(request, createWriteStream(path));
  return {status: 204};
};
```

我们不需要检查文件是否存在，如果存在，只需覆盖即可。我们再次使用 `pipe` 来将可读流中的数据移动到可写流中，在本例中是将请求的数据移动到文件中。但是由于 `pipe` 没有为返回 `Promise` 而编写，所以我们必须编写包装器 `pipeStream` ，它从调用 `pipe` 的结果中创建一个 `Promise` 。

当打开文件 `createWriteStream` 时出现问题时仍然会返回一个流，但是这个流会触发 `'error'` 事件。例如，如果网络出现故障，请求的输出流也可能失败。所以我们连接两个流的 `'error'` 事件来拒绝 `Promise` 。当 `pipe` 完成时，它会关闭输出流，从而导致触发 `'finish'` 事件。这是我们可以成功解析 `Promise` 的地方（不返回任何内容）。

完整的服务器脚本请见 `eloquentjavascript.net/code/file_server.js` 。读者可以下载该脚本，并且在安装依赖项之后，使用 Node 启动你自己的文件服务器。当然你可以修改并扩展该脚本，来完成本章的习题或进行实验。

命令行工具 `curl` 在类 Unix 系统（比如 Mac 或者 Linux）中得到广泛使用，可用于产生 HTTP 请求。接下来的会话用于简单测试我们的服务器。这里需要注意， `-X` 用于设置请求方法， `-d` 用于包含请求正文。

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d hello http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
hello
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

由于`file.txt`一开始不存在，因此第一请求失败。而`PUT`请求则创建文件，因此我们看到下一个请求可以成功获取该文件。在使用`DELETE`请求删除该文件后，第三次`GET`请求再次找不到该文件。

本章小结

Node 是一个不错的小型系统，可让我们在非浏览器环境下运行 JavaScript。Node 最初的设计意图是完成网络任务，扮演网络中的节点。但同时也能用来执行任何脚本任务，如果你觉得编写 JavaScript 代码是一件惬意的事情，那么使用 Node 来自动完成每天的任务是非常不错的。

NPM 为你所能想到的功能（当然还有相当多你想不到的）提供了包，你可以通过使用`npm`程序，获取并安装这些包。Node 也附带了许多内建模块，包括`fs`模块（处理文件系统）、`http`模块（执行 HTTP 服务器并生成 HTTP 请求）。

Node 中的所有输入输出都是异步的，除非你明确使用函数的同步变体，比如`readFileSync`。当调用异步函数时，使用者提供回调，并且 Node 会在准备好的时候，使用错误值和结果（如果有的话）调用它们。

习题

搜索工具

在 Unix 系统上，有一个名为`grep`的命令行工具，可以用来在文件中快速搜索正则表达式。

编写一个可以从命令行运行的 Node 脚本，其行为类似`grep`。 它将其第一个命令行参数视为正则表达式，并将任何其他参数视为要搜索的文件。 它应该输出内容与正则表达式匹配的，任何文件的名称。

当它有效时，将其扩展，以便当其中一个参数是目录时，它将搜索该目录及其子目录中的所有文件。

按照你认为合适的方式，使用异步或同步文件系统函数。 配置一些东西，以便同时请求多个异步操作可能会加快速度，但不是很大，因为大多数文件系统一次只能读取一个东西。

目录创建

尽管我们的文件服务器中的`DELETE`方法可以删除目录（使用`rmdir`），但服务器目前不提供任何方法来创建目录。

添加对`MKCOL`方法（“make column”）的支持，它应该通过调用`fs`模块的`mkdir`创建一个目录。 `MKCOL`并不是广泛使用的 HTTP 方法，但是它在 WebDAV 标准中有相同的用途，这个标准在 HTTP 之上规定了一组适用于创建文档的约定。

你可以使用实现`DELETE`方法的函数，作为`MKCOL`方法的蓝图。 当找不到文件时，尝试用`mkdir`创建一个目录。 当路径中存在目录时，可以返回 204 响应，以便目录创建请求是幂等的。 如果这里存在非目录文件，则返回错误代码。 代码 400（“Bad Request”，请求无效）是适当的。

网络上的公共空间

由于文件服务器提供了任何类型的文件服务，甚至只要包含正确的`Content-Type`协议头，你可以使用其提供网站服务。由于该服务允许每个人删除或替换文件，因此这是一类非常有趣的网站：任何人只要使用正确的 HTTP 请求，都可以修改、改进并破坏文件。但这仍然是一个网站。

请编写一个基础的 HTML 页面，包含一个简单的 JavaScript 文件。将该文件放在文件服务器的数据目录下，并在你的浏览器中打开这些文件。

接下来，作为进阶练习或是周末作业，将你迄今为止在本书中学习到的内容整合起来，构建一个对用户友好的界面，在网站内部修改网站。

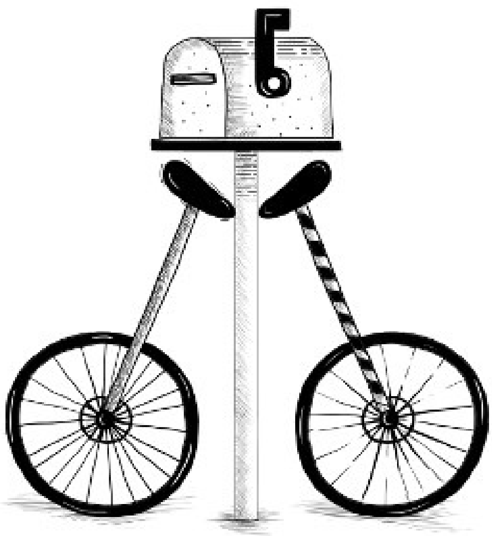
使用 HTML 表单编辑组成网站的文件内容，允许用户使用 HTTP 请求在服务器上更新它们，如第十八章所述。

刚开始的时候，该页面仅允许用户编辑单个文件，然后进行修改，允许选择想要编辑的文件。向文件服务器发送请求时，若URL是一个目录，服务器会返回该目录下的文件列表，你可以利用该特性实现你的网页。

不要直接编辑文件服务器开放的代码，如果你犯了什么错误，很有可能就破坏了你的代码。相反，将你的代码保存在公共访问目录之外，测试时再将其拷贝到公共目录中。

二十一、项目：技能分享网站

> 原文：Project: Skill-Sharing Website
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译
>
> 部分参考了《JavaScript 编程精解（第 2 版）》
>
> If you have knowledge, let others light their candles at it.
>
> Margaret Fuller



技能分享会是一个活动，其中兴趣相同的人聚在一起，针对他们所知的事情进行小型非正式的展示。在园艺技能分享会上，可以解释如何耕作芹菜。如果在编程技能分享小组中，你可以顺便给每个人讲讲 Node.js。

在计算机领域中，这类聚会往往名为用户小组，是开阔眼界、了解行业新动态或仅仅接触兴趣相同的人的好方法。许多大城市都会有 JavaScript 聚会。这类聚会往往是可以免费参加的，而且我发现我参加过的那些聚会都非常友好热情。

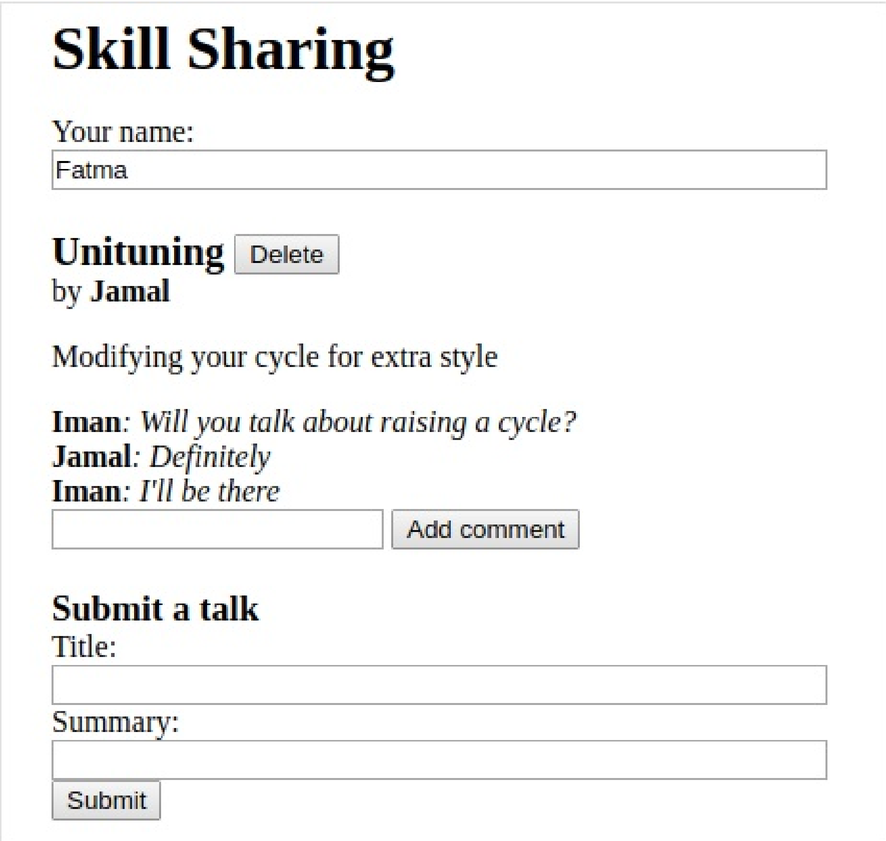
在最后的项目章节中，我们的目标是建立网站，管理特定技能分享会的讨论内容。假设一个小组的人会在成员办公室中定期举办关于独轮车的聚会。上一个组织者搬到了另一个城市，并且没人可以站出来接下来他的任务。我们需要一个系统，让参与者可以在系统中发言并相互讨论，这样就不需要一个中心组织人员了。

就像上一章一样，本章中的一些代码是为 Node.js 编写的，并且直接在你正在查看的 HTML 页面中运行它不太可行。 该项目的完整代码可以从`eloquentjavascript.net/code/skillsharing.zip`下载。

设计

本项目的服务器部分为 Node.js 编写，客户端部分则为浏览器编写。服务器存储系统数据并将其提供给客户端。它也提供实现客户端系统的文件。

服务器保存了为下次聚会提出的对话列表。每个对话包括参与人员姓名、标题和该对话的相关评论。客户端允许用户提出新的对话（将对话添加到列表中）、删除对话和评论已存在的对话。每当用户做了修改时，客户端会向服务器发送关于更改的 HTTP 请求。



我们创建应用来展示一个实时视图，来展示目前已经提出的对话和评论。每当某些人在某些地点提交了新的对话或添加新评论时，所有在浏览器中打开页面的人都应该立即看到变化。这个特性略有挑战，网络服务器无法建立到客户端的连接，也没有好方法来知道有哪些客户端现在在查看特定网站。

该问题的一个解决方案叫作长时间轮询，这恰巧是 Node 的设计动机之一。

长轮询

为了能够立即提示客户端某些信息发生了改变，我们需要建立到客户端的连接。由于通常浏览器无法接受连接，而且客户端通常在路由后面，它无论如何都会拒绝这类连接，因此由服务器初始化连接是不切实际的。

我们可以安排客户端来打开连接并保持该连接，因此服务器可以使用该连接在必要时传送信息。

但 HTTP 请求只是简单的信息流：客户端发送请求，服务器返回一条响应，就是这样。有一种名为 WebSocket 的技术，受到现代浏览器的支持，是的我们可以建立连接并进行任意的数据交换。但如何正确运用这项技术是较为复杂的。

本章我们将会使用一种相对简单的技术：长轮询（Long Polling）。客户端会连续使用定时的 HTTP 请求向服务器询问新信息，而当没有新信息需要报告时服务器会简单地推迟响应。

只要客户端确保其可以持续不断地建立轮询请求，就可以在信息可用之后，从服务器快速地接收到信息。例如，若 Fatma 在浏览器中打开了技能分享程序，浏览器会发送请求询问是否有更新，且等待请求的响应。当 Iman 在自己的浏览器中提交了关于"极限降滑独轮车"的对话之后。服务器发现 Fatma 在等待更新请求，并将新的对话作为响应发送给待处理的请求。Fatma 的浏览器将会接收到数据并更新屏幕展示对话内容。

为了防止连接超时（因为连接一定时间不活跃后会被中断），长轮询技术常常为每个请求设置一个最大等待时间，只要超过了这个时间，即使没人有任何需要报告的信息也会返回响应，在此之后，客户端会建立一个新的请求。定期重新发送请求也使得这种技术更具鲁棒性，允许客户端从临时的连接失败或服务器问题中恢复。

使用了长轮询技术的繁忙的服务器，可以有成百上千个等待的请求，因此也就有这么多个 TCP 连接处于打开状态。Node简化了多连接的管理工作，而不是建立单独线程来控制每个连接，这对这样的系统是非常合适的。

HTTP 接口

在我们设计服务器或客户端的代码之前，让我们先来思考一下两者均会涉及的一点：双方通信的 HTTP 接口。

我们会使用 JSON 作为请求和响应正文的格式，就像第二十章中的文件服务器一样，我们尝试充分利用 HTTP 方法。所有接口均以 `/talks` 路径为中心。不以 `/talks` 开头的路径则用于提供静态文件服务，即用于实现客户端系统的 HTML 和 JavaScript 代码。

访问 `/talks` 的 `GET` 请求会返回如下所示的 JSON 文档。

```
[{"title": "Unituning",
  "presenter": "Jamal",
  "summary": "Modifying your cycle for extra style",
  "comment": []}]
```

我们可以发送 `PUT` 请求到类似于 `/talks/Unituning` 之类的 URL 上来创建新对话，在第二个斜杠后的那部分是对话的名称。`PUT` 请求正文应当包含一个 JSON 对象，其中有一个 `presenter` 属性和一个 `summary` 属性。

因为对话标题可以包含空格和其他无法正常出现在 URL 中的字符，因此我们必须使用 `encodeURIComponent` 函数来编码标题字符串，并构建 URL。

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

下面这个请求用于创建关于“空转”的对话。

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{"presenter": "Maureen",
 "summary": "Standing still on a unicycle"}
```

我们也可以使用 `GET` 请求通过这些 URL 获取对话的 JSON 数据，或使用 `DELETE` 请求通过这些 URL 删除对话。

为了在对话中添加一条评论，可以向诸如 `/talks/Unituning/comments` 的 URL 发送 `POST` 请求，JSON 正文包含 `author` 属性和 `message` 属性。

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72

{"author": "Iman",
 "message": "Will you talk about raising a cycle?"}
```

为了支持长轮询，如果没有新的信息可用，发送到 `/talks` 的 `GET` 请求可能会包含额外的标题，通知服务器延迟响应。 我们将使用通常用于管理缓存的一对协议头： `ETag` 和 `If-None-Match` 。

服务器可能在响应中包含 `ETag` （“实体标签”）协议头。 它的值是标识资源当前版本的字符串。 当客户稍后再次请求该资源时，可以通过包含一个 `If-None-Match` 头来进行条件请求，该头的值保存相同的字符串。 如果资源没有改变，服务器将响应状态码 304，这意味着“未修改”，告诉客户端它的缓存版本仍然是最新的。 当标签与服务器不匹配时，服务器正常响应。

我们需要这样的东西，通过它客户端可以告诉服务器它有哪个版本的对话列表，仅当列表发生变化时，服务器才会响应。 但服务器不是立即返回 304 响应，它应该停止响应，并且仅当有新东西的可用，或已经过去了给定的时间时才返回。 为了将长轮询请求与常规条件请求区分开来，我们给他们另一个标头 `Prefer: wait=90` ，告诉服务器客户端最多等待 90 秒的响应。

服务器将保留版本号，每次对话更改时更新，并将其用作`ETag`值。 客户端可以在对话变更时通知此类要求：

```
GET /talks HTTP/1.1
If-None-Match: "4"
Prefer: wait=90

(time passes)

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "5"
Content-Length: 295

[....]
```

这里描述的协议并没有任何访问控制。每个人都可以评论、修改对话或删除对话。因为因特网中充满了流氓，因此将这类没有进一步保护的系统放在网络上最后可能并不是很好。

服务器

让我们开始构建程序的服务器部分。本节的代码可以在 Node.js 中执行。

路由

我们的服务器会使用`createServer`来启动 HTTP 服务器。在处理新请求的函数中，我们必须区分我们支持的请求的类型（根据方法和路径确定）。我们可以使用一长串的`if`语句完成该任务，但还存在一种更优雅的方式。

路由可以作为帮助把请求调度传给能处理该请求的函数。路径匹配正则表达式`/^\/talks\/([^\/]+)$/`（`/talks/`带着对话名称）的`PUT`请求，应当由指定函数处理。此外，路由可以帮助我们提取路径中有意义的部分，在本例中会将对话的标题（包裹在正则表达式的括号之中）传递给处理器函数。

在 NPM 中有许多优秀的路由包，但这里我们自己编写一个路由来展示其原理。

这里给出`router.js`，我们随后将在服务器模块中使用`require`获取该模块。

```
const {parse} = require("url");

module.exports = class Router {
  constructor() {
    this.routes = [];
  }
  add(method, url, handler) {
    this.routes.push({method, url, handler});
  }
  resolve(context, request) {
    let path = parse(request.url).pathname;

    for (let {method, url, handler} of this.routes) {
      let match = url.exec(path);
      if (!match || request.method != method) continue;
      let urlParts = match.slice(1).map(decodeURIComponent);
      return handler(context, ...urlParts, request);
    }
    return null;
  }
};
```

该模块导出`Router`类。我们可以使用路由对象的`add`方法来注册一个新的处理器，并使用`resolve`方法解析请求。

找到处理器之后，后者会返回一个响应，否则为`null`。它会逐个尝试路由（根据定义顺序排序），当找到一个匹配的路由时返回`true`。

路由会使用`context`值调用处理器函数（这里是服务器实例），将请求对象中的字符串，与已定义分组中的正则表达式匹配。传递给处理器的字符串必须进行 URL 解码，因为原始 URL 中可能包含`%20`风格的代码。

文件服务

当请求无法匹配路由中定义的任何请求类型时，服务器必须将其解释为请求位于`public`目录下的某个文件。服务器可以使用第二十章中定义的文件服务器来提供文件服务，但我们并不需要也不想对文件支持 PUT 和 DELETE 请求，且我们想支持类似于缓存等高级特性。因此让我们使用 NPM 中更为可靠且经过充分测试的静态文件服务器。

我选择了`ecstatic`。它并不是 NPM 中唯一的此类服务，但它能够完美工作且符合我们的意图。`ecstatic`模块导出了一个函数，我们可以调用该函数，并传递一个配置对象来生成一个请求处理函数。我们使用`root`选项告知服务器文件搜索位置。

```
const {createServer} = require("http");
const Router = require("./router");
const ecstatic = require("ecstatic");

const router = new Router();
const defaultHeaders = {"Content-Type": "text/plain"};

class SkillShareServer {
  constructor(talks) {
    this.talks = talks;
    this.version = 0;
    this.waiting = [];

    let fileServer = ecstatic({root: "./public"});
    this.server = createServer((request, response) => {
      let resolved = router.resolve(this, request);
      if (resolved) {
        resolved.catch(error => {
          if (error.status != null) return error;
          return {body: String(error), status: 500};
        }).then(({body,
                  status = 200,
                  headers = defaultHeaders}) => {
          response.writeHead(status, headers);
          response.end(body);
        });
      } else {
        fileServer(request, response);
      }
    });
  }
  start(port) {
    this.server.listen(port);
  }
  stop() {
    this.server.close();
  }
}
```

它使用上一章中的文件服务器的类似约定来处理响应 - 处理器返回 `Promise`，可解析为描述响应的对象。 它将服务器包装在一个对象中，它也维护它的状态。

作为资源的对话

已提出的对话存储在服务器的 `talks` 属性中，这是一个对象，属性名称是对话标题。这些对话会展现为 `/talks/[title]` 下的 HTTP 资源，因此我们需要将处理器添加我们的路由中供客户端选择，来实现不同的方法。

获取（`GET`）单个对话的请求处理器，必须查找对话并使用对话的 JSON 数据作为响应，若不存在则返回 404 错误响应码。

```
const talkPath = /^\/talks\/([^\/]+)$/;

router.add("GET", talkPath, async (server, title) => {
  if (title in server.talks) {
    return {body: JSON.stringify(server.talks[title]),
            headers: {"Content-Type": "application/json"}};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

删除对话时，将其从`talks`对象中删除即可。

```
router.add("DELETE", talkPath, async (server, title) => {
  if (title in server.talks) {
    delete server.talks[title];
    server.updated();
  }
  return {status: 204};
});
```

我们将在稍后定义`updated`方法，它通知等待有关更改的长轮询请求。

为了获取请求正文的内容，我们定义一个名为`readStream`的函数，从可读流中读取所有内容，并返回解析为字符串的`Promise`。

```
function readStream(stream) {
  return new Promise((resolve, reject) => {
    let data = "";
    stream.on("error", reject);
    stream.on("data", chunk => data += chunk.toString());
    stream.on("end", () => resolve(data));
  });
}
```

需要读取响应正文的函数是`PUT`的处理器，用户使用它创建新对话。该函数需要检查数据中是否有`presenter`和`summary`属性，这些属性都是字符串。任何来自外部的数据都可能是无意义的，我们不希望错误请求到达时会破坏我们的内部数据模型，或者导致服务崩溃。

若数据看起来合法，处理器会将对话转化为对象，存储在`talks`对象中，如果有标题相同的对话存在则覆盖，并再次调用`updated`。

```
router.add("PUT", talkPath,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let talk;
  try { talk = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!talk ||
      typeof talk.presenter != "string" ||
      typeof talk.summary != "string") {
    return {status: 400, body: "Bad talk data"};
  }
  server.talks[title] = {title,
                         presenter: talk.presenter,
                         summary: talk.summary,
                         comments: []};
  server.updated();
  return {status: 204};
});
```

在对话中添加评论也是类似的。我们使用`readStream`来获取请求内容，验证请求数据，若看上去合法，则将其存储为评论。

```
router.add("POST", /^\/talks\/([^\/]+)\/comments$/,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let comment;
  try { comment = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!comment ||
      typeof comment.author != "string" ||
      typeof comment.message != "string") {
    return {status: 400, body: "Bad comment data"};
  } else if (title in server.talks) {
    server.talks[title].comments.push(comment);
    server.updated();
    return {status: 204};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

尝试向不存在的对话中添加评论会返回 404 错误。

长轮询支持

服务器中最值得探讨的方面是处理长轮询的部分代码。当 URL 为`/talks`的`GET`请求到来时，它可能是一个常规请求或一个长轮询请求。

我们可能在很多地方，将对话列表发送给客户端，因此我们首先定义一个简单的辅助函数，它构建这样一个数组，并在响应中包含`ETag`协议头。

```
SkillShareServer.prototype.talkResponse = function() {
  let talks = [];
  for (let title of Object.keys(this.talks)) {
    talks.push(this.talks[title]);
  }
  return {
    body: JSON.stringify(talks),
    headers: {"Content-Type": "application/json",
              "ETag": `"${this.version}"`}
  };
};
```

处理器本身需要查看请求头，来查看是否存在 `If-None-Match` 和 `Prefer` 标头。Node 在其小写名称下存储协议头，根据规定其名称是不区分大小写的。

```
router.add("GET", /^\/talks$/, async (server, request) => {
  let tag = /"(.*)"/.exec(request.headers["if-none-match"]);
  let wait = /\bwait=(\d+)/.exec(request.headers["prefer"]);
  if (!tag || tag[1] != server.version) {
    return server.talkResponse();
  } else if (!wait) {
    return {status: 304};
  } else {
    return server.waitForChanges(Number(wait[1]));
  }
});
```

如果没有给出标签，或者给出的标签与服务器的当前版本不匹配，则处理器使用对话列表来响应。如果请求是有条件的，并且对话没有变化，我们查阅 `Prefer` 标题来查看，是否应该延迟响应或立即响应。

用于延迟请求的回调函数存储在服务器的 `waiting` 数组中，以便在发生事件时通知它们。`waitForChanges` 方法也会立即设置一个定时器，当请求等待了足够长时，以 304 状态来响应。

```
SkillShareServer.prototype.waitForChanges = function(time) {
  return new Promise(resolve => {
    this.waiting.push(resolve);
    setTimeout(() => {
      if (!this.waiting.includes(resolve)) return;
      this.waiting = this.waiting.filter(r => r != resolve);
      resolve({status: 304});
    }, time * 1000);
  });
};
```

使用 `updated` 注册一个更改，会增加 `version` 属性并唤醒所有等待的请求。

```
var changes = [];

SkillShareServer.prototype.updated = function() {
  this.version++;
  let response = this.talkResponse();
  this.waiting.forEach(resolve => resolve(response));
  this.waiting = [];
};
```

服务器代码这样就完成了。 如果我们创建一个`SkillShareServer`的实例，并在端口 8000 上启动它，那么生成的 HTTP 服务器，将服务于`public`子目录中的文件，以及`/ talks` URL 下的一个对话管理界面。

```
new SkillShareServer(Object.create(null)).start(8000);
```

客户端

技能分享网站的客户端部分由三个文件组成：微型 HTML 页面、样式表以及 JavaScript 文件。

HTML

在网络服务器提供文件服务时，有一种广为使用的约定是：当请求直接访问与目录对应的路径时，返回名为`index.html`的文件。我们使用的文件服务模块`ecstatic`就支持这种约定。当请求路径为/时，服务器会搜索文件`./public/index.html`（`./public`是我们赋予的根目录），若文件存在则返回文件。

因此，若我们希望浏览器指向我们服务器时展示某个特定页面，我们将其放在`public/index.html`中。这就是我们的`index`文件。

```
<!doctype html>
<meta charset="utf-8">
<title>Skill Sharing</title>
<link rel="stylesheet" href="skillsharing.css">

<h1>Skill Sharing</h1>

<script src="skillsharing_client.js"></script>
```

它定义了文档标题并包含一个样式表，除了其它东西，它定义了几种样式，确保对话之间有一定的空间。

最后，它在页面顶部添加标题，并加载包含客户端应用的脚本。

动作

应用状态由对话列表和用户名称组成，我们将它存储在一个`{talks, user}`对象中。 我们不允许用户界面直接操作状态或发送 HTTP 请求。 反之，它可能会触发动作，它描述用户正在尝试做什么。

```
function handleAction(state, action) {
  if (action.type == "setUser") {
    localStorage.setItem("userName", action.user);
    return Object.assign({}, state, {user: action.user});
  } else if (action.type == "setTalks") {
    return Object.assign({}, state, {talks: action.talks});
  } else if (action.type == "newTalk") {
    fetchOK(talkURL(action.title), {
      method: "PUT",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        presenter: state.user,
        summary: action.summary
      })
    }).catch(reportError);
  } else if (action.type == "deleteTalk") {
    fetchOK(talkURL(action.talk), {method: "DELETE"})
      .catch(reportError);
  } else if (action.type == "newComment") {
    fetchOK(talkURL(action.talk) + "/comments", {
      method: "POST",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        author: state.user,
        message: action.message
      })
    }).catch(reportError);
  }
  return state;
}
```

我们将用户的名字存储在`localStorage`中，以便在页面加载时恢复。

需要涉及服务器的操作使用`fetch`，将网络请求发送到前面描述的 HTTP 接口。 我们使用包装函数`fetchOK`，它确保当服务器返回错误代码时，拒绝返回的`Promise`。

```
function fetchOK(url, options) {
  return fetch(url, options).then(response => {
    if (response.status < 400) return response;
    else throw new Error(response.statusText);
  });
}
```

这个辅助函数用于为某个对话，使用给定标题建立 URL。

```
function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}
```

当请求失败时，我们不希望我们的页面丝毫不变，不给予任何提示。因此我们定义一个函数，名为`reportError`，至少在发生错误时向用户展示一个对话框。

```
function reportError(error) {
  alert(String(error));
}
```

渲染组件

我们将使用一个方法，类似于我们在第十九章中所见，将应用拆分为组件。但由于某些组件不需要更新，或者在更新时总是完全重新绘制，所以我们不将它们定义为类，而是直接返回 DOM 节点的函数。例如，下面是一个组件，显示用户可以向它输入名称的字段的：

```
function renderUserField(name, dispatch) {
  return elt("label", {}, "Your name: ", elt("input", {
    type: "text",
    value: name,
    onchange(event) {
      dispatch({type: "setUser", user: event.target.value});
    }
  }));
}
```

用于构建 DOM 元素的 `elt` 函数是我们在第十九章中使用的函数。

类似的函数用于渲染对话，包括评论列表和添加新评论的表单。

```
function renderTalk(talk, dispatch) {
  return elt(
    "section", {className: "talk"},
    elt("h2", null, talk.title, " ", elt("button", {
      type: "button",
      onclick() {
        dispatch({type: "deleteTalk", talk: talk.title});
      }
    }, "Delete")),
    elt("div", null, "by ",
        elt("strong", null, talk.presenter)),
    elt("p", null, talk.summary),
    ...talk.comments.map(renderComment),
    elt("form", {
      onsubmit(event) {
        event.preventDefault();
        let form = event.target;
        dispatch({type: "newComment",
                  talk: talk.title,
                  message: form.elements.comment.value});
        form.reset();
      }
    }, elt("input", {type: "text", name: "comment"}), " ",
       elt("button", {type: "submit"}, "Add comment")));
}
```

`submit` 事件处理器调用 `form.reset`，在创建 `"newComment"` 动作后清除表单的内容。

在创建适度复杂的 DOM 片段时，这种编程风格开始显得相当混乱。有一个广泛使用的（非标准的）JavaScript 扩展叫做 JSX，它允许你直接在你的脚本中编写 HTML，这可以使这样的代码更漂亮（取决于你认为漂亮是什么）。在实际运行这种代码之前，必须在脚本上运行一个程序，将伪 HTML 转换为 JavaScript 函数调用，就像我们在这里用的东西。

评论更容易渲染。

```
function renderComment(comment) {
  return elt("p", {className: "comment"},
             elt("strong", null, comment.author),
             ": ", comment.message);
}
```

最后，用户可以使用表单创建新对话，它渲染为这样。

```
function renderTalkForm(dispatch) {
  let title = elt("input", {type: "text"});
  let summary = elt("input", {type: "text"});
  return elt("form", {
    onsubmit(event) {
      event.preventDefault();
      dispatch({type: "newTalk",
                title: title.value,
                summary: summary.value});
      event.target.reset();
    }
  }, elt("h3", null, "Submit a Talk"),
     elt("label", null, "Title: ", title),
     elt("label", null, "Summary: ", summary),
     elt("button", {type: "submit"}, "Submit"));
}
```

轮询

为了启动应用，我们需要对话的当前列表。 由于初始加载与长轮询过程密切相关 -- 轮询时必须使用来自加载的 `ETag` -- 我们将编写一个函数来不断轮询服务器的 `/ talks` ，并且在新的对话集可用时，调用回调函数。

```
async function pollTalks(update) {
  let tag = undefined;
  for (;;) {
    let response;
    try {
      response = await fetchOK("/talks", {
        headers: tag && {"If-None-Match": tag,
                         "Prefer": "wait=90"}
      });
    } catch (e) {
      console.log("Request failed: " + e);
      await new Promise(resolve => setTimeout(resolve, 500));
      continue;
    }
    if (response.status == 304) continue;
    tag = response.headers.get("ETag");
    update(await response.json());
  }
}
```

这是一个 `async` 函数，因此循环和等待请求更容易。 它运行一个无限循环，每次迭代中，通常检索对话列表。或者，如果这不是第一个请求，则带有使其成为长轮询请求的协议头。

当请求失败时，函数会等待一会儿，然后再次尝试。 这样，如果你的网络连接断了一段时间然后又恢复，应用可以恢复并继续更新。 通过`setTimeout`解析的`Promise`，是强制`async`函数等待的方法。

当服务器回复 304 响应时，这意味着长轮询请求超时，所以函数应该立即启动下一个请求。 如果响应是普通的 200 响应，它的正文将当做 JSON 而读取并传递给回调函数，并且它的`ETag`协议头的值为下一次迭代而存储。

应用

以下组件将整个用户界面结合在一起。

```
class SkillShareApp {
  constructor(state, dispatch) {
    this.dispatch = dispatch;
    this.talkDOM = elt("div", {className: "talks"});
    this.dom = elt("div", null,
                   renderUserField(state.user, dispatch),
                   this.talkDOM,
                   renderTalkForm(dispatch));
    this.setState(state);
  }

  setState(state) {
    if (state.talks != this.talks) {
      this.talkDOM.textContent = "";
      for (let talk of state.talks) {
        this.talkDOM.appendChild(
          renderTalk(talk, this.dispatch));
      }
      this.talks = state.talks;
    }
  }
}
```

当对话改变时，这个组件重新绘制所有这些组件。 这很简单，但也是浪费。 我们将在练习中回顾一下。

我们可以像这样启动应用：

```
function runApp() {
  let user = localStorage.getItem("userName") || "Anon";
  let state, app;
  function dispatch(action) {
    state = handleAction(state, action);
    app.setState(state);
  }

  pollTalks(talks => {
    if (!app) {
      state = {user, talks};
      app = new SkillShareApp(state, dispatch);
      document.body.appendChild(app.dom);
    } else {
      dispatch({type: "setTalks", talks});
    }
  }).catch(reportError);
}

runApp();
```

若你执行服务器并同时为`localhost:8000/`打开两个浏览器窗口，你可以看到在一个窗口中执行动作时，另一个窗口中会立即做出反应。

习题

下面的习题涉及修改本章中定义的系统。为了使用该系统进行工作，请确保首先下载代码，安装了 Node，并使用`npm install`安装了项目的所有依赖。

磁盘持久化

技能分享服务只将数据存储在内存中。这就意味着当服务崩溃或以为任何原因重启时，所有的对话和评论都会丢失。

扩展服务使得其将对话数据存储到磁盘上，并在程序重启时自动重新加载数据。不要担心效率，只要用最简单的代码让其可以工作即可。

重置评论字段

由于我们常常无法在 DOM 节点中找到唯一替换的位置，因此整批地重绘对话是个很好的工作机制。但这里有个例外，若你开始在对话的评论字段中输入一些文字，而在另一个窗口向同一条对话添加了一条评论，那么第一个窗口中的字段就会被重绘，会移除掉其内容和焦点。

在激烈的讨论中，多人同时添加评论，这将是非常烦人的。 你能想出办法解决它吗？